# RxJava Internals

## Agenda

- Introduction
- Observables and Observer
- Operators
- Combining Observables
- Multicasting, Replaying and Caching
- Concurrency
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

# Agenda

- **Introduction**
- Observables and Observer
- Operators
- Combining Observables
- Multicasting, Replaying and Caching
- Concurrency
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

## Introduction:version

- RxJava 1.x, 2.x, or 3.0
  - RxJava 3.0:
    - This version will continue to grow and receive new features, while
  - RxJava 1.x:
    - Has not been developed further since March 21, 2018,
  - RxJava 2.x:
    - Will be maintained for bug fixes only until February 28, 2021. However, there are other considerations that may lead you to use RxJava 1.x or 2.x.

## Introduction:version

- RxJava 1.x, 2.x, or 3.0
  - Legacy Code
  - In RxJava, there are **multiple libraries** that can be used to make several Java APIs reactive and plug into RxJava seamlessly.
    - RxJava-JDBC, RxAndroid, RxJava-Extras, RxNetty, and RxJavaFX.
    - 2020:Only RxAndroid and RxJavaFX have been fully ported to RxJava 2.x, and many other libraries will follow
    - 2021: majority of them have been ported to RxJava 2.x and some to RxJava 3.0, too.

# Introduction:when

- What circumstances warrant a reactive approach?
  - Always?
    - when you first start out, is yes! You always want to take a reactive approach.
  - Occasionally there will be times when a reactive approach may not be optimal.
    - As experienced reactive programmer it will be clear when to return a List<String> instead of an Observable<String>

## Agenda

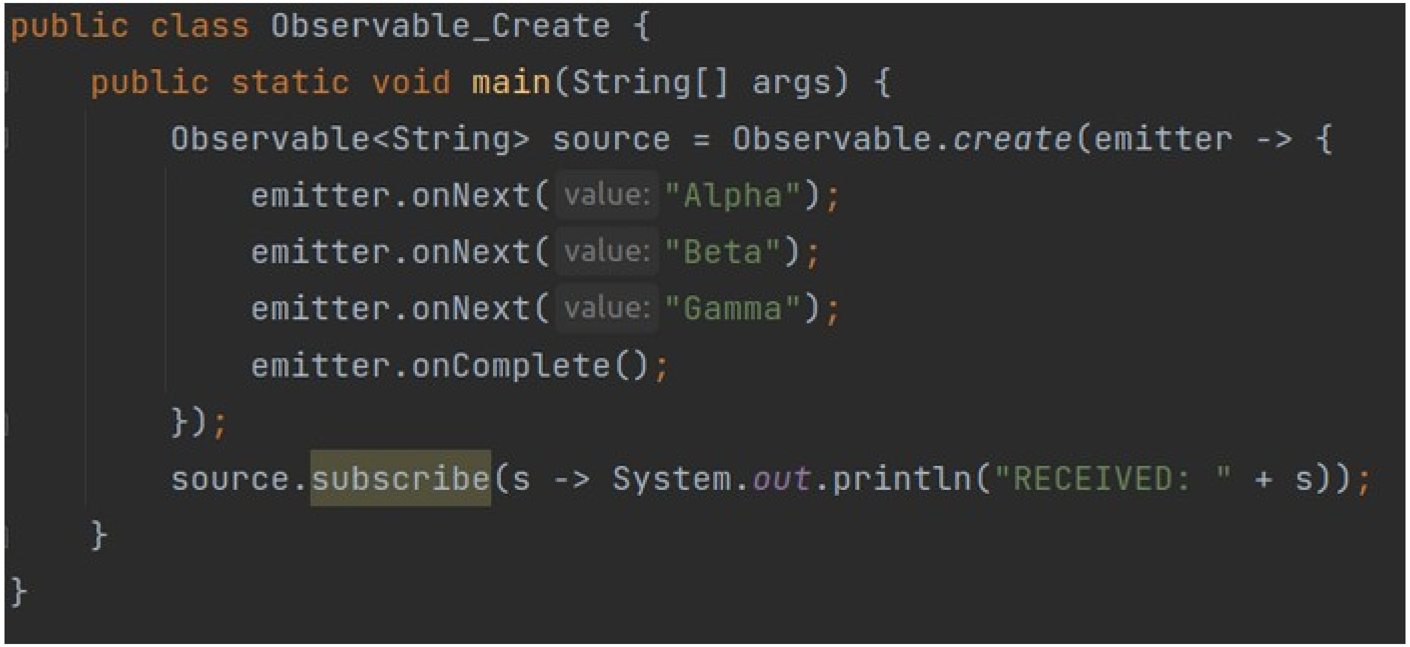
- Introduction
- **Observables and Observer**
- Operators
- Combining Observables
- Multicasting, Replaying and Caching
- Concurrency
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

## Observables and Observer:Factories

- At the highest level, an Observable works by passing three types of events:
  - onNext(): This passes each item one at a time from the source Observable all the way down to the Observer.
  - onComplete(): This communicates a completion event all the way down to the Observer, indicating that no more onNext() calls will occur.
  - onError(): This communicates an error down the chain to the Observer, which typically defines how to handle it. Unless a retry() operator is used to intercept the error, the Observable chain typically terminates, and no more emissions occur.

## Observables and Observer:Factories:Create:ex-1

- A source Observable is an Observable from where emissions originate.

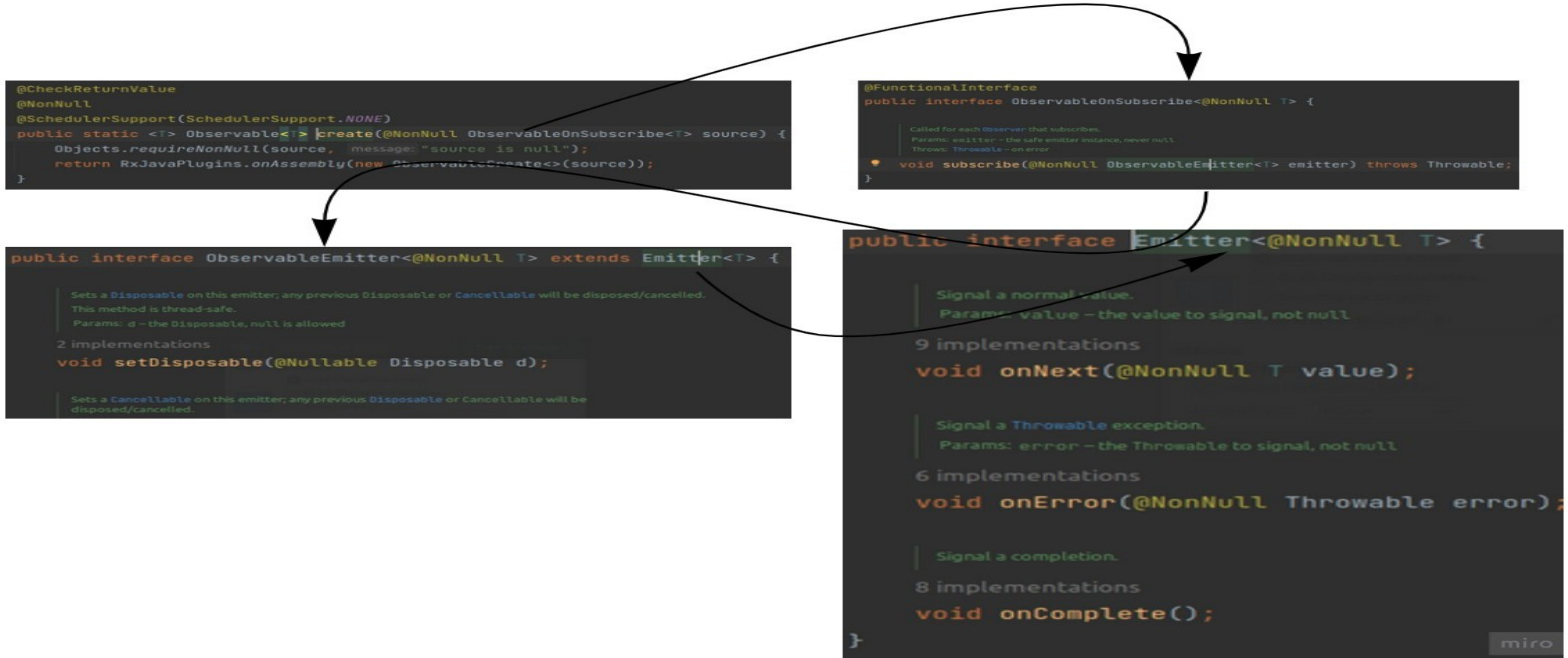


```
public class Observable_Create {
    public static void main(String[] args) {
        Observable<String> source = Observable.create(emitter -> {
            emitter.onNext( value: "Alpha");
            emitter.onNext( value: "Beta");
            emitter.onNext( value: "Gamma");
            emitter.onComplete();
        });
        source.subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

```
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Gamma
```

## Observables and Observer:Factories:Create:ex-1

- A source Observable is an Observable from where emissions originate.



```java
@CheckReturnValue
@NonNull
@SchedulerSupport(SchedulerSupport.NONE)
public static <T> Observable<T> create(@NonNull ObservableOnSubscribe<T> source) {
    Objects.requireNonNull(source,  message: "source is null");
    return RxJavaPlugins.onAssembly(new ObservableCreate<>(source));
}
```

```java
@FunctionalInterface
public interface ObservableOnSubscribe<@NonNull T> {

        Called for each Observer that subscribes.
        Params: emitter – the safe emitter instance, never null
        Throws: Throwable – on error
    void subscribe(@NonNull ObservableEmitter<T> emitter) throws Throwable;
}
```

```java
public interface ObservableEmitter<@NonNull T> extends Emitter<T> {

        Sets a Disposable on this emitter; any previous Disposable or Cancellable will be disposed/cancelled.
        This method is thread-safe.
        Params: d – the Disposable, null is allowed
    2 implementations
    void setDisposable(@Nullable Disposable d);

        Sets a Cancellable on this emitter; any previous Disposable or Cancellable will be
        disposed/cancelled.
```

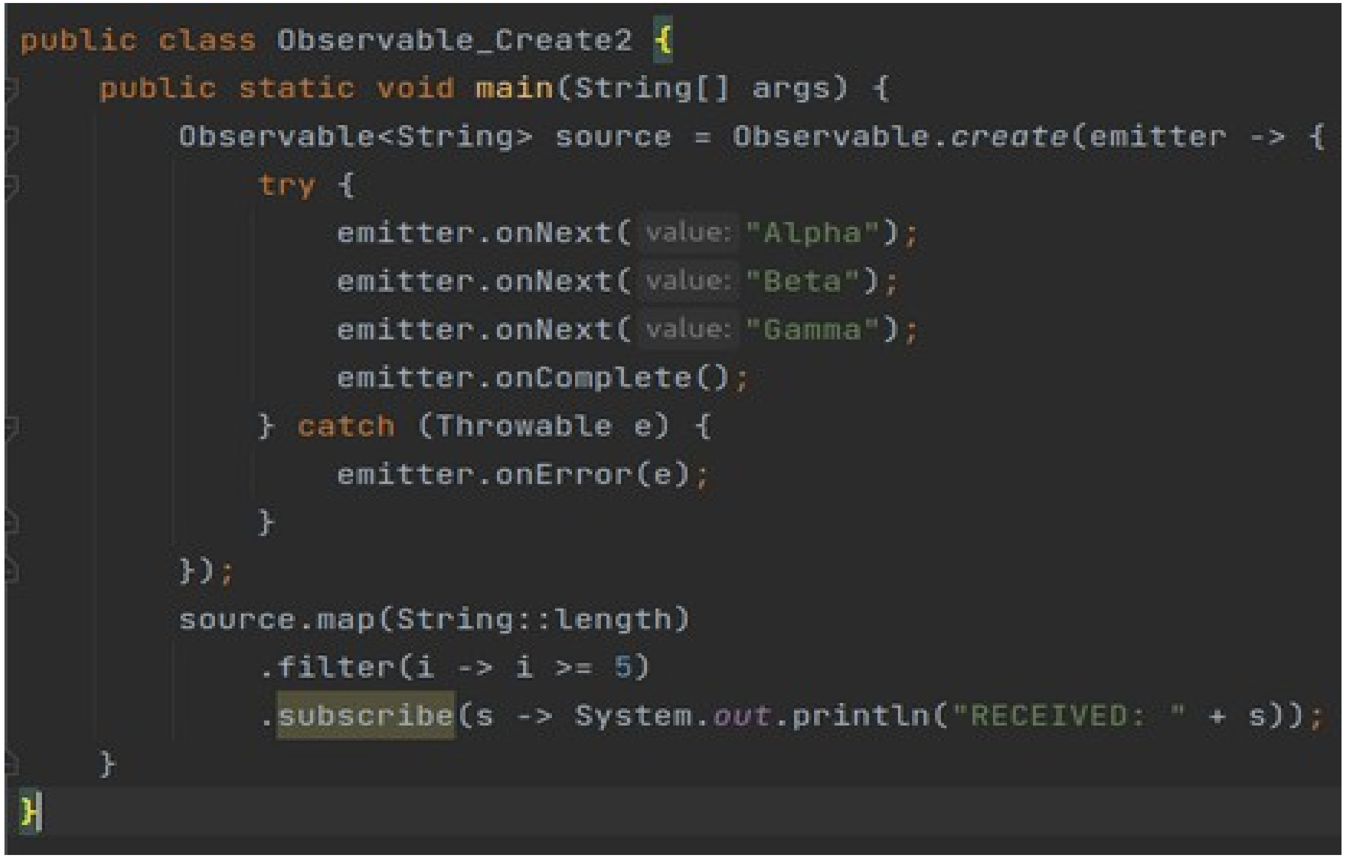
```java
public interface Emitter<@NonNull T> {

        Signal a normal value.
        Params: value – the value to signal, not null
    9 implementations
    void onNext(@NonNull T value);

        Signal a Throwable exception.
        Params: error – the Throwable to signal, not null
    6 implementations
    void onError(@NonNull Throwable error);

        Signal a completion.
    8 implementations
    void onComplete();
}
```

## Observables and Observer:Factories:Create:ex-1

- A source Observable is an Observable from where emissions originate.
  - Actually, the create() method accepts as a parameter an object of the ObservableOnSubscribe type that has only one method, subscribe(ObservableEmitter emitter), which accepts an ObservableEmitter type, which, in turn, extends the Emitter interface that has three methods: onNext(), onError(), and onComplete().

# Observables and Observer:Factories:Create

- A source Observable is an Observable from where emissions originate.

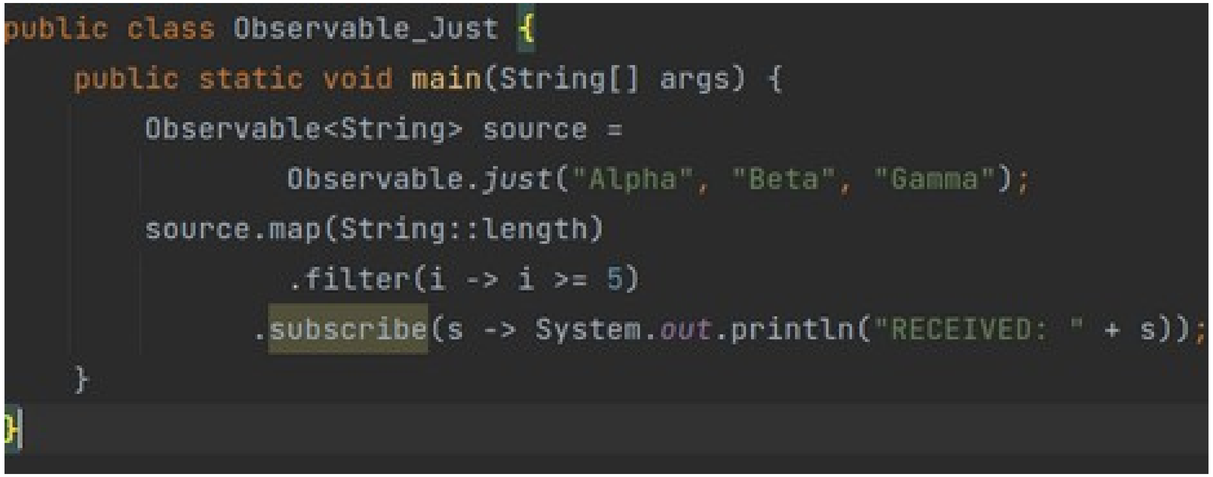


```java
public class Observable_Create2 {
    public static void main(String[] args) {
        Observable<String> source = Observable.create(emitter -> {
            try {
                emitter.onNext( value: "Alpha");
                emitter.onNext( value: "Beta");
                emitter.onNext( value: "Gamma");
                emitter.onComplete();
            } catch (Throwable e) {
                emitter.onError(e);
            }
        });
        source.map(String::length)
            .filter(i -> i >= 5)
            .subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

```
RECEIVED: 5
RECEIVED: 5
```

## Observables and Observer:Factories:Just

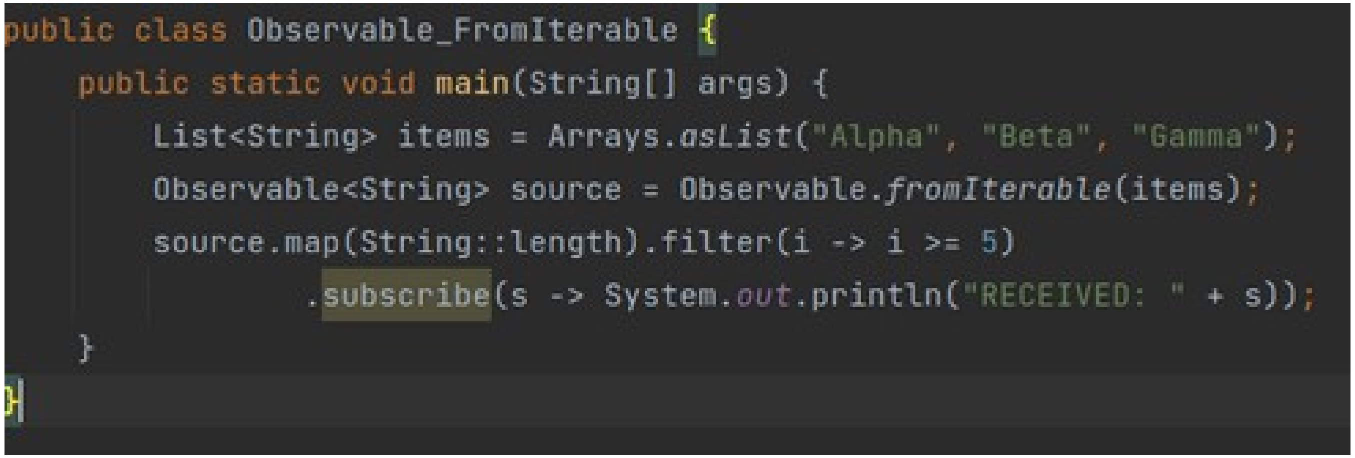
- The create(...) is a generic Observable factory method for creating an Observable in which you will explicitly dictate how values are passed to the Subscriber
  - However, if you want just to transform a single value into the Observable the just() is probably more readable option to choose. just() takes an argument and sends it as next and then it sends completed right after the next.



```
public class Observable_Just {
    public static void main(String[] args) {
        Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma");
        source.map(String::length)
                .filter(i -> i >= 5)
              .subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

## Observables and Observer:Factories:fromIterable

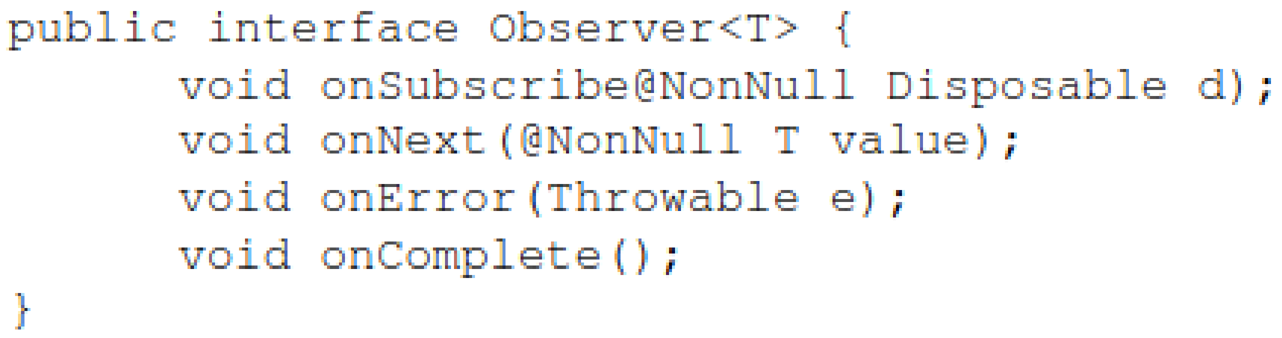
- A source Observable is an Observable from where emissions originate.
  - Observable.fromIterable() to emit the items from any Iterabletype, such as a List, for example.
  - Frequently used since Iterable in Java is used often and can easily be made reactive:



```
public class Observable_FromIterable {
    public static void main(String[] args) {
        List<String> items = Arrays.asList("Alpha", "Beta", "Gamma");
        Observable<String> source = Observable.fromIterable(items);
        source.map(String::length).filter(i -> i >= 5)
                .subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

## Observables and Observer:Observer

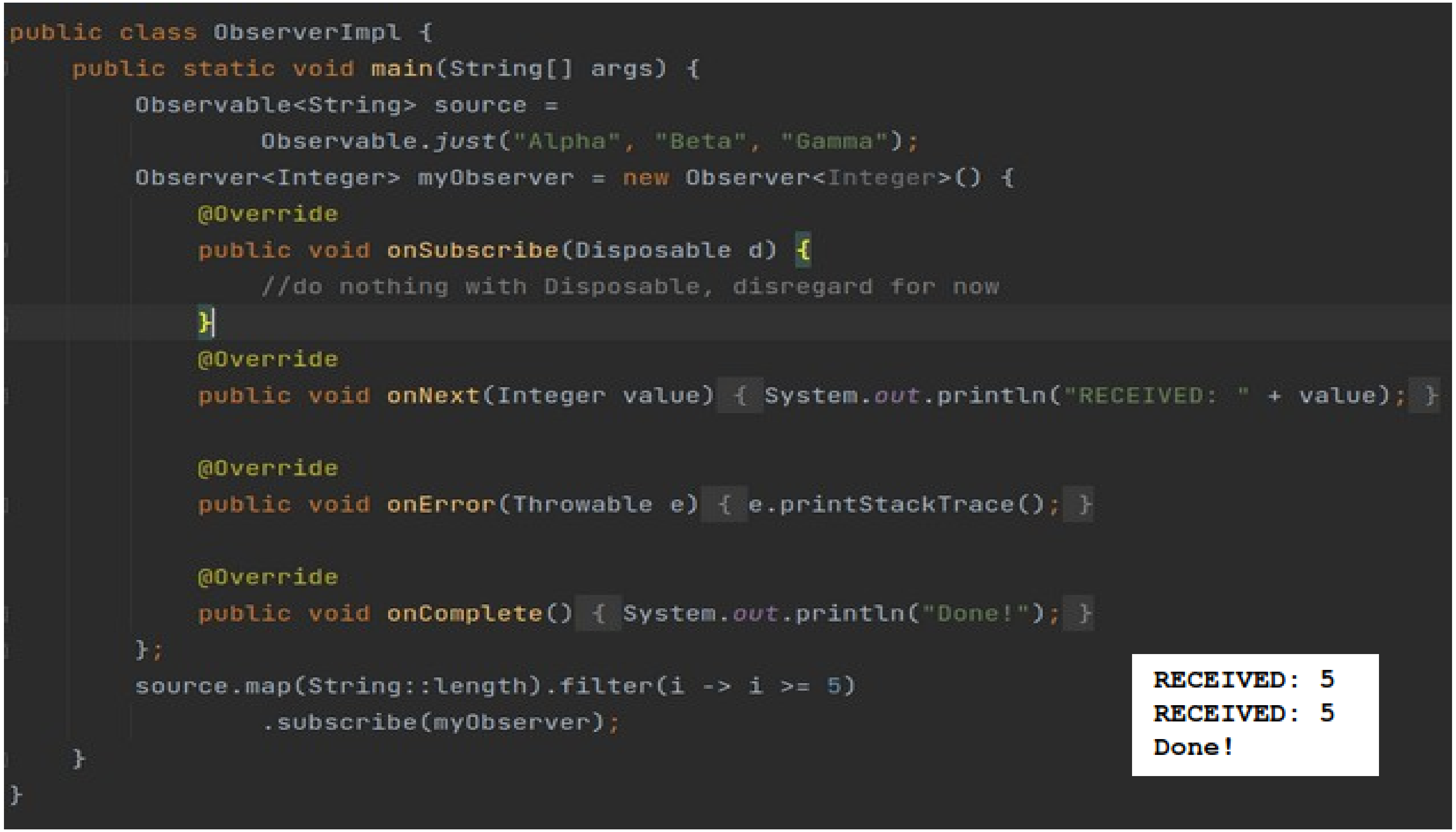
- The Observer interface
  - The onNext(), onComplete(), and onError() methods actually compose the Observer type – an interface implemented throughout RxJava to communicate the corresponding events.



```
public interface Observer<T> {
      void onSubscribe@NonNull Disposable d);
      void onNext(@NonNull T value);
      void onError(Throwable e);
      void onComplete();
}
```

## Observables and Observer:Observer

- The Observer interface:Impl

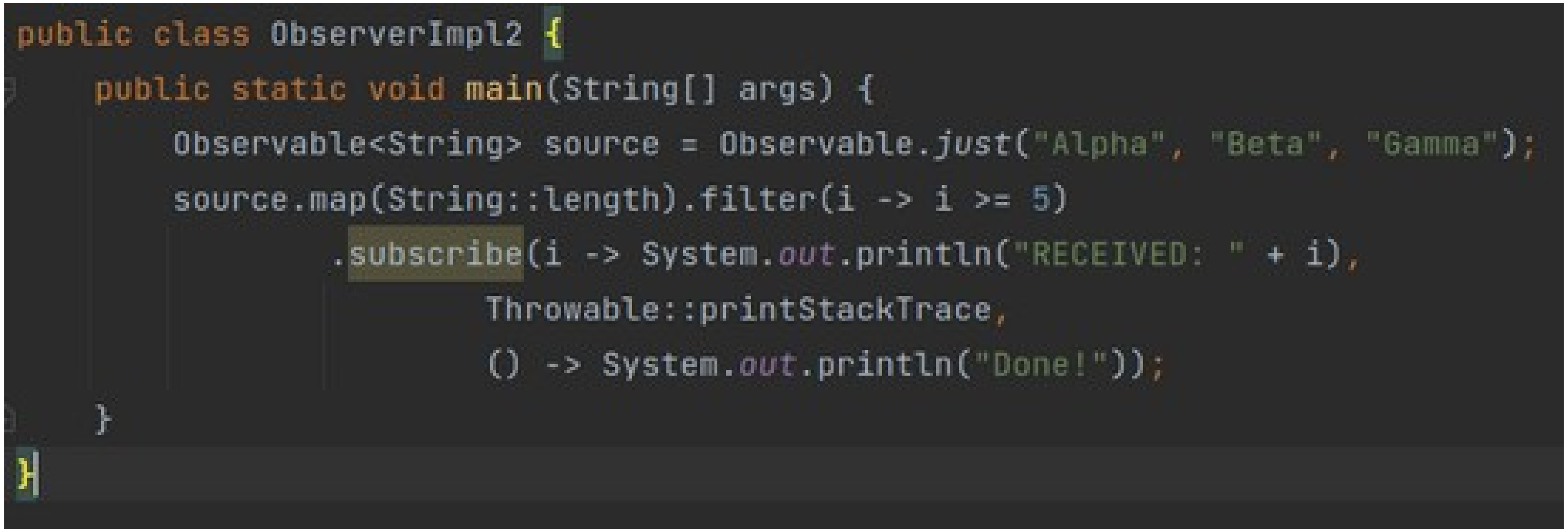


```
public class ObserverImpl {
    public static void main(String[] args) {
        Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma");
        Observer<Integer> myObserver = new Observer<Integer>() {
            @Override
            public void onSubscribe(Disposable d) {
                //do nothing with Disposable, disregard for now
            }
            @Override
            public void onNext(Integer value) { System.out.println("RECEIVED: " + value); }

            @Override
            public void onError(Throwable e) { e.printStackTrace(); }

            @Override
            public void onComplete() { System.out.println("Done!"); }
        };
        source.map(String::length).filter(i -> i >= 5)
                .subscribe(myObserver);
    }
}
```

```
RECEIVED: 5
RECEIVED: 5
Done!
```

# Observables and Observer:Observer

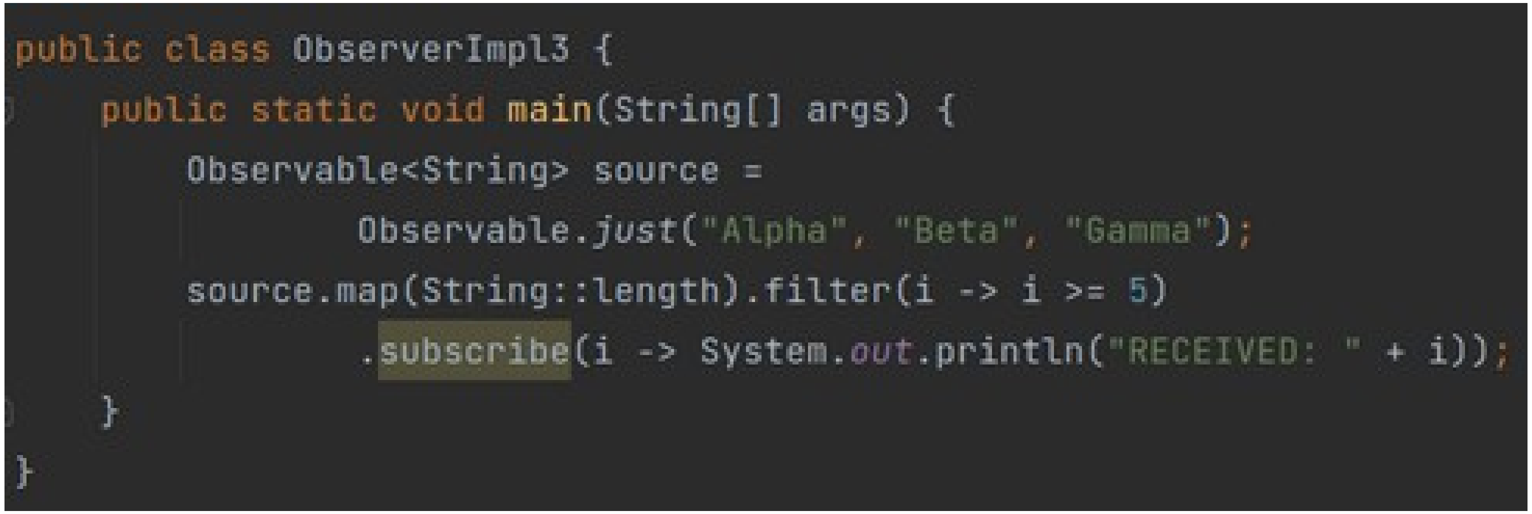
- The Observer interface:Impl:
  - Implementing Observer is a bit verbose and cumbersome.
  - Thankfully, the **subscribe() method is overloaded** to accept lambda arguments for our three events.



```java
public class ObserverImpl2 {
    public static void main(String[] args) {
        Observable<String> source = Observable.just("Alpha", "Beta", "Gamma");
        source.map(String::length).filter(i -> i >= 5)
                .subscribe(i -> System.out.println("RECEIVED: " + i),
                        Throwable::printStackTrace,
                        () -> System.out.println("Done!"));
    }
}
```

## Observables and Observer:Observer

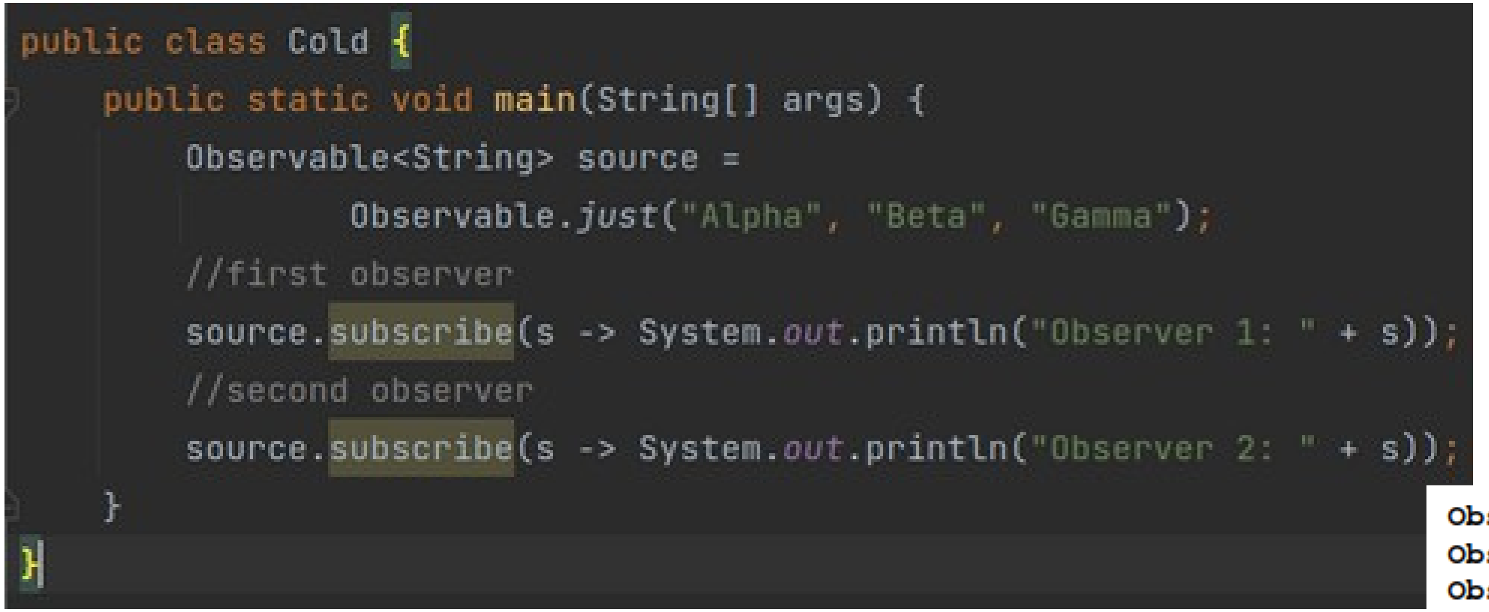
- The Observer interface:Impl:
  - Implementing Observer is a bit verbose and cumbersome.
  - You can even omit **onError** and just specify onNext:



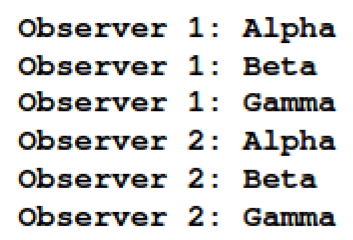
```
public class ObserverImpl3 {
    public static void main(String[] args) {
        Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma");
        source.map(String::length).filter(i -> i >= 5)
                .subscribe(i -> System.out.println("RECEIVED: " + i));
    }
}
```

## Observables and Observer:Cold

- A cold Observable is much like a music CD that is provided to each listener, so each person can hear all the tracks any time they start listening to it. In the same manner, a cold Observable replays the emissions to each Observer, ensuring that it gets all the data.



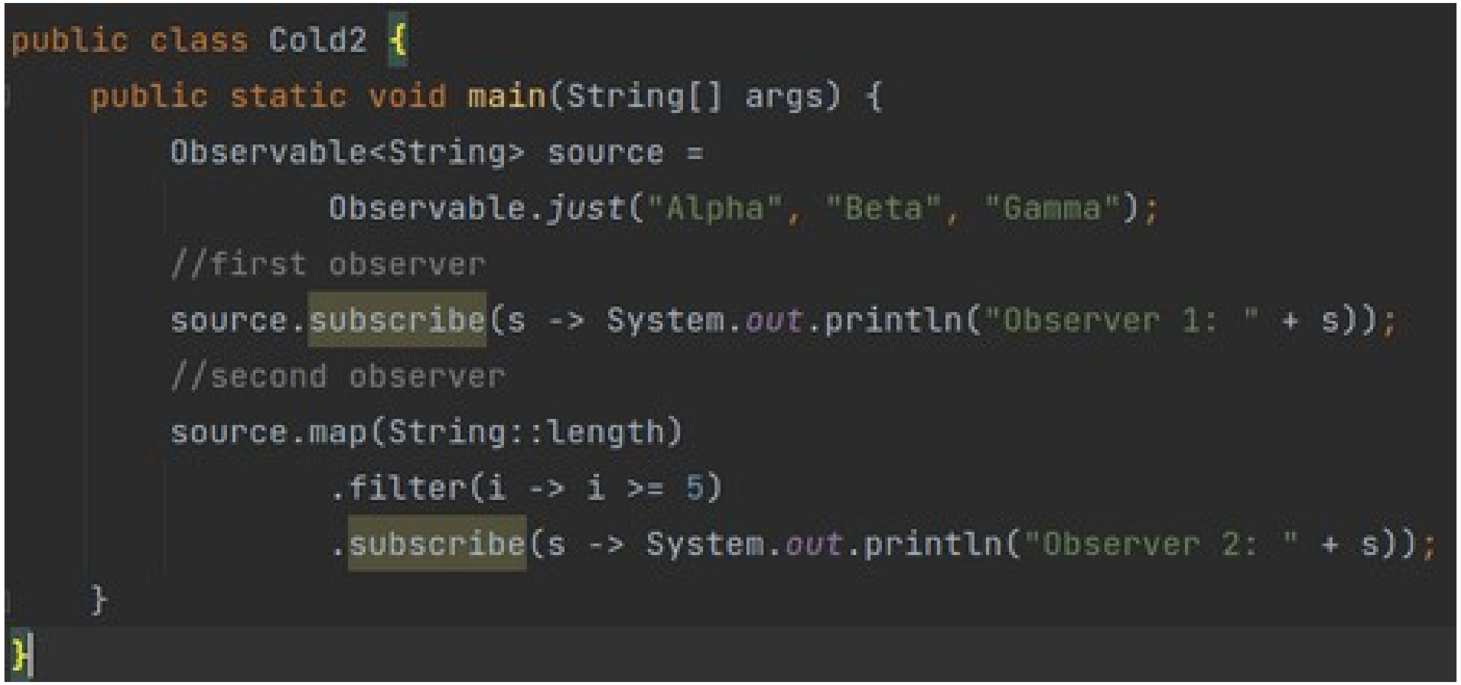
```
public class Cold {
    public static void main(String[] args) {
        Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma");
        //first observer
        source.subscribe(s -> System.out.println("Observer 1: " + s));
        //second observer
        source.subscribe(s -> System.out.println("Observer 2: " + s));
    }
}
```



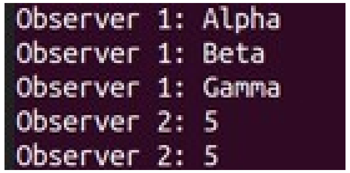
```
Observer 1: Alpha
Observer 1: Beta
Observer 1: Gamma
Observer 2: Alpha
Observer 2: Beta
Observer 2: Gamma
```

# Observables and Observer:Cold

- Even if the second Observer transforms its emissions with operators, it will still get its own stream of emissions. Using operators such as map() and filter() against a cold Observable preserves the cold nature of the produced observables:

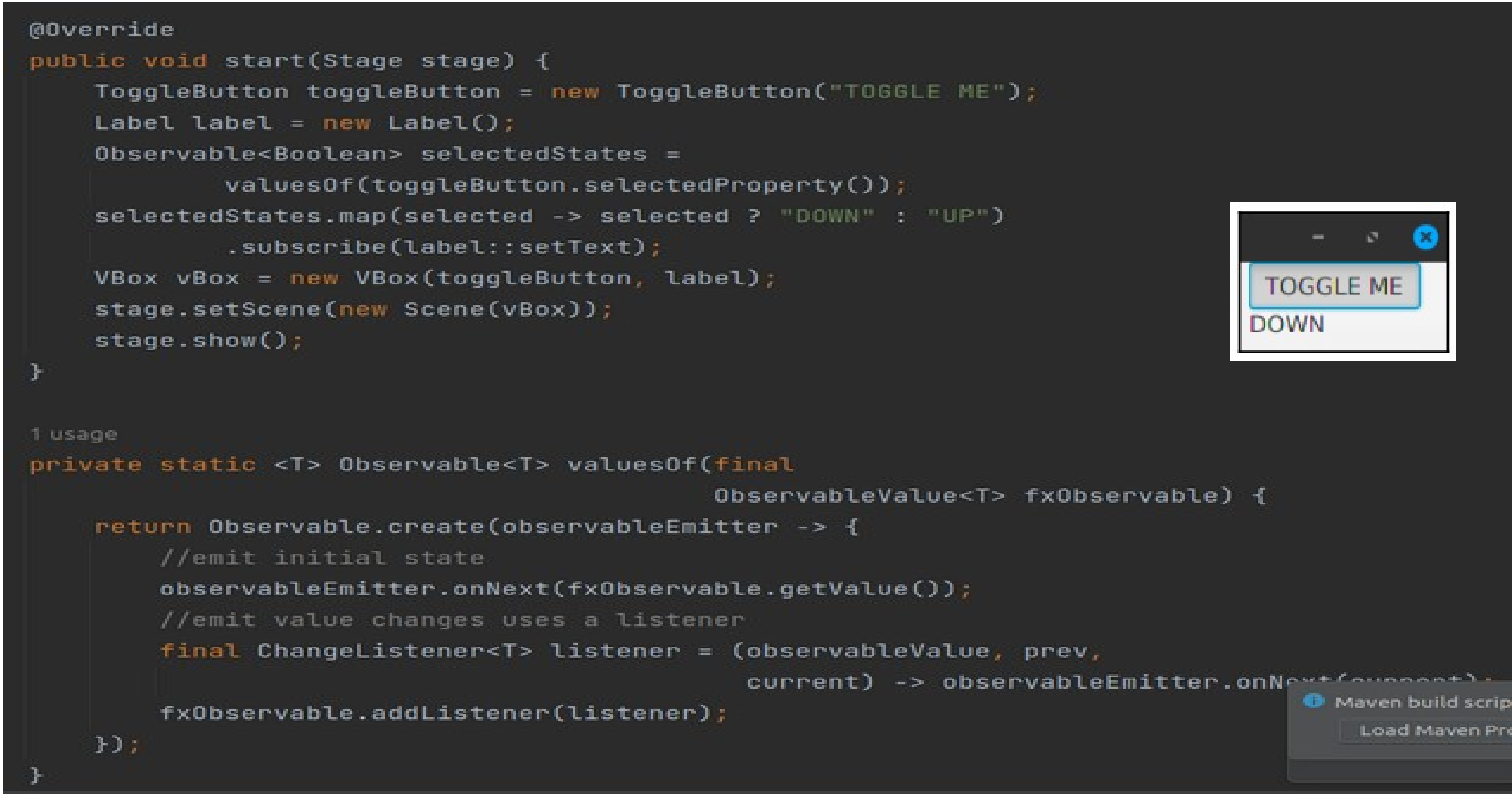


```java
public class Cold2 {
    public static void main(String[] args) {
        Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma");
        //first observer
        source.subscribe(s -> System.out.println("Observer 1: " + s));
        //second observer
        source.map(String::length)
                .filter(i -> i >= 5)
                .subscribe(s -> System.out.println("Observer 2: " + s));
    }
}
```

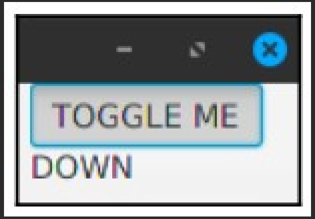


```
Observer 1: Alpha
Observer 1: Beta
Observer 1: Gamma
Observer 2: 5
Observer 2: 5
```

# Observables and Observer:Hot



```
@Override
public void start(Stage stage) {
    ToggleButton toggleButton = new ToggleButton("TOGGLE ME");
    Label label = new Label();
    Observable<Boolean> selectedStates =
            valuesOf(toggleButton.selectedProperty());
    selectedStates.map(selected -> selected ? "DOWN" : "UP")
            .subscribe(label::setText);
    VBox vBox = new VBox(toggleButton, label);
    stage.setScene(new Scene(vBox));
    stage.show();
}

1 usage
private static <T> Observable<T> valuesOf(final
                                          ObservableValue<T> fxObservable) {
    return Observable.create(observableEmitter -> {
        //emit initial state
        observableEmitter.onNext(fxObservable.getValue());
        //emit value changes uses a listener
        final ChangeListener<T> listener = (observableValue, prev,
                                            current) -> observableEmitter.onN
        fxObservable.addListener(listener);
    });
}
```



## Observables and Observer:Hot

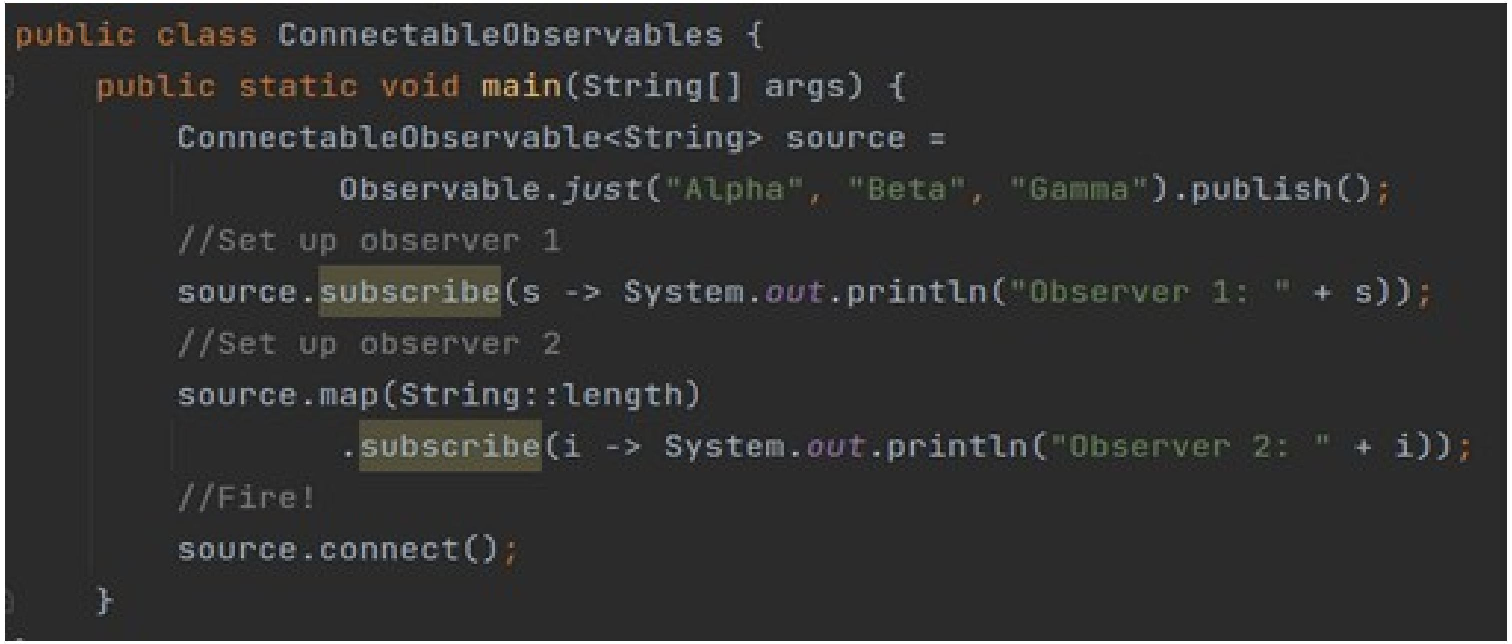
- A hot Observable is more like a radio station.
  - It broadcasts the same emissions to all observers at the same time.
  - If an Observer subscribes to a hot Observable, receives some emissions, and then another Observer subscribes later, that second Observer will have missed those emissions.
    - Just like a radio station, if you tune in too late, you will have missed that song.

# Observables and Observer:Hot

- A hot Observable is more like a radio station.
  - Every time you click TOGGLE ME, the ToggleButton is invoked and the Observable<Boolean> emits a true or false value that switches the selection state.
  - This is a simple example, showing that this Observable is emitting events, but is also emitting data in the form of true or false.
  - It transforms that boolean value into a String object and forces an Observer object to modify the text of the Label.
  - We only have one **Observer** in this JavaFX example.
    - If we were to **add more observers** to listen to the ToggleButton events after emissions have occurred, those new observers would have **missed earlier emissions**.

## Observables and Observer:ConnectableObservables

- A helpful form of a hot Observable is the ConnectableObservable class. It takes any Observable, even if it is **cold, and makes it hot** so that all emissions are played to all observers at once.
- To do this conversion, you simply need to call **publish()** on an Observable, and it will yield a ConnectableObservable object.



```java
public class ConnectableObservables {
    public static void main(String[] args) {
        ConnectableObservable<String> source =
                Observable.just("Alpha", "Beta", "Gamma").publish();
        //Set up observer 1
        source.subscribe(s -> System.out.println("Observer 1: " + s));
        //Set up observer 2
        source.map(String::length)
                .subscribe(i -> System.out.println("Observer 2: " + i));
        //Fire!
        source.connect();
    }
```

```
Observer 1: Alpha
Observer 2: 5
Observer 1: Beta
Observer 2: 4
Observer 1: Gamma
Observer 2: 5
```

## Observables and Observer:ConnectableObservables

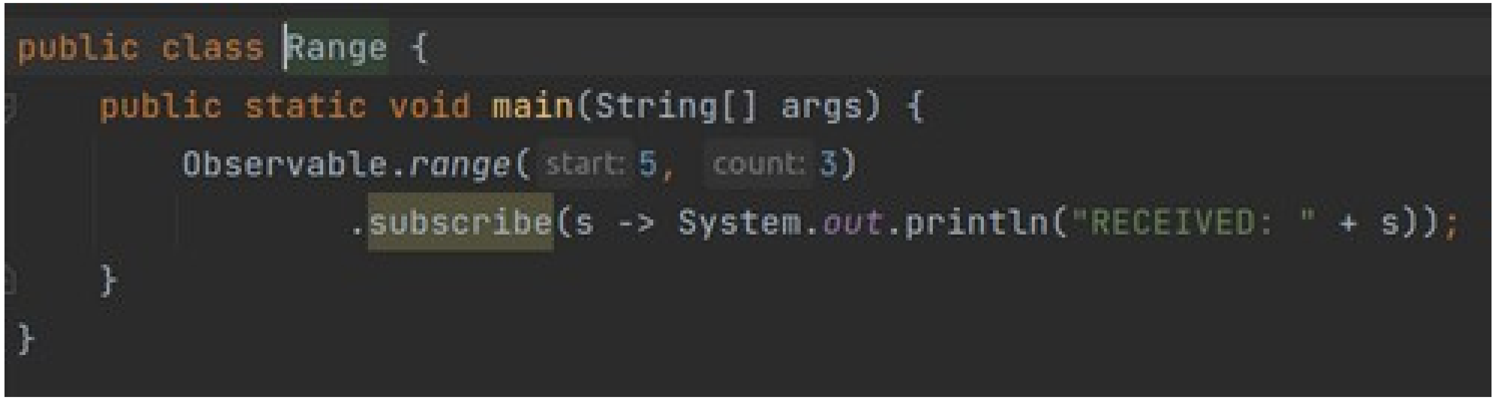
- But note that subscribing does not start the emission. You need to call the **connect()** method on the ConnectableObservable object to start it.
  - This allows you to set up all your observers first, before the first value is emitted.
  - Using ConnectableObservable to force each emission to go to all observers is known as **multicasting**
  - The ConnectableObservable is helpful in **preventing the replaying** of data to each Observer.
    - You may want to do this when the **replaying is expensive** and you decide that emissions should go to all observers at the same time

## Observables and Observer:ConnectableObservables

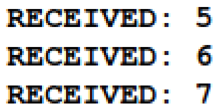
- Multiple observers normally result in **multiple stream instances upstream**.
  - But using **publish() to return ConnectableObservable** consolidates all the upstream operations into a **single stream**

## Observables and Observer:Range

- Note closely that the two arguments for Observable.range() are not lower and upper bounds.
  - The first argument is the initial value.
  - The second argument is the total count of emissions, which will include both the initial value and subsequent incremented values.



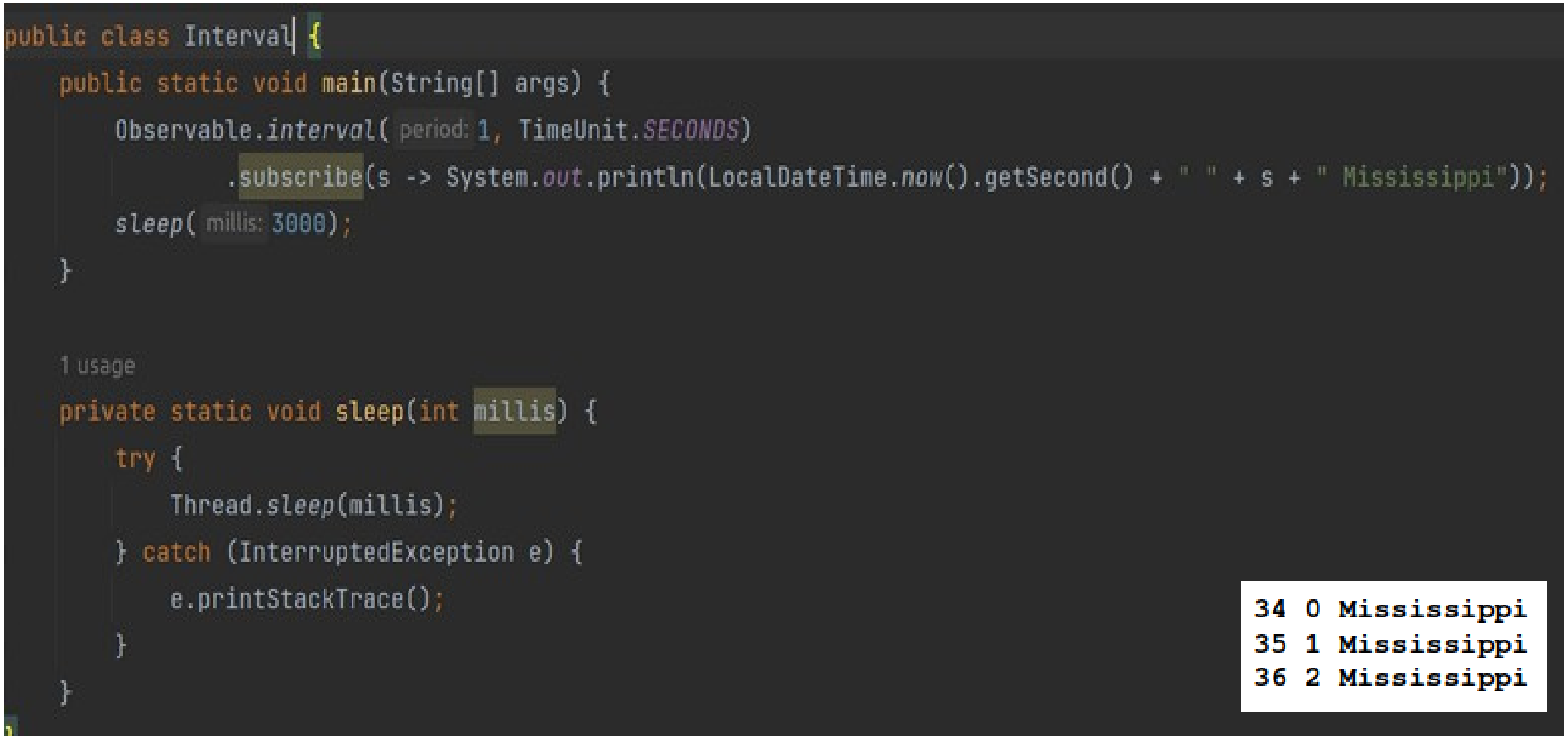
```
public class Range {
    public static void main(String[] args) {
        Observable.range( start: 5,  count: 3)
                .subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```



```
RECEIVED: 5
RECEIVED: 6
RECEIVED: 7
```

# Observables and Observer:Interval

- Interval emits consecutive long values (starting at 0) with the specified time interval between emissions. Here, we have an Observable<Long> that emits every second:



```java
public class Interval {
    public static void main(String[] args) {
        Observable.interval( period: 1, TimeUnit.SECONDS)
                .subscribe(s -> System.out.println(LocalDateTime.now().getSecond() + " " + s + " Mississippi"));
        sleep( millis: 3000);
    }

    1 usage
    private static void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

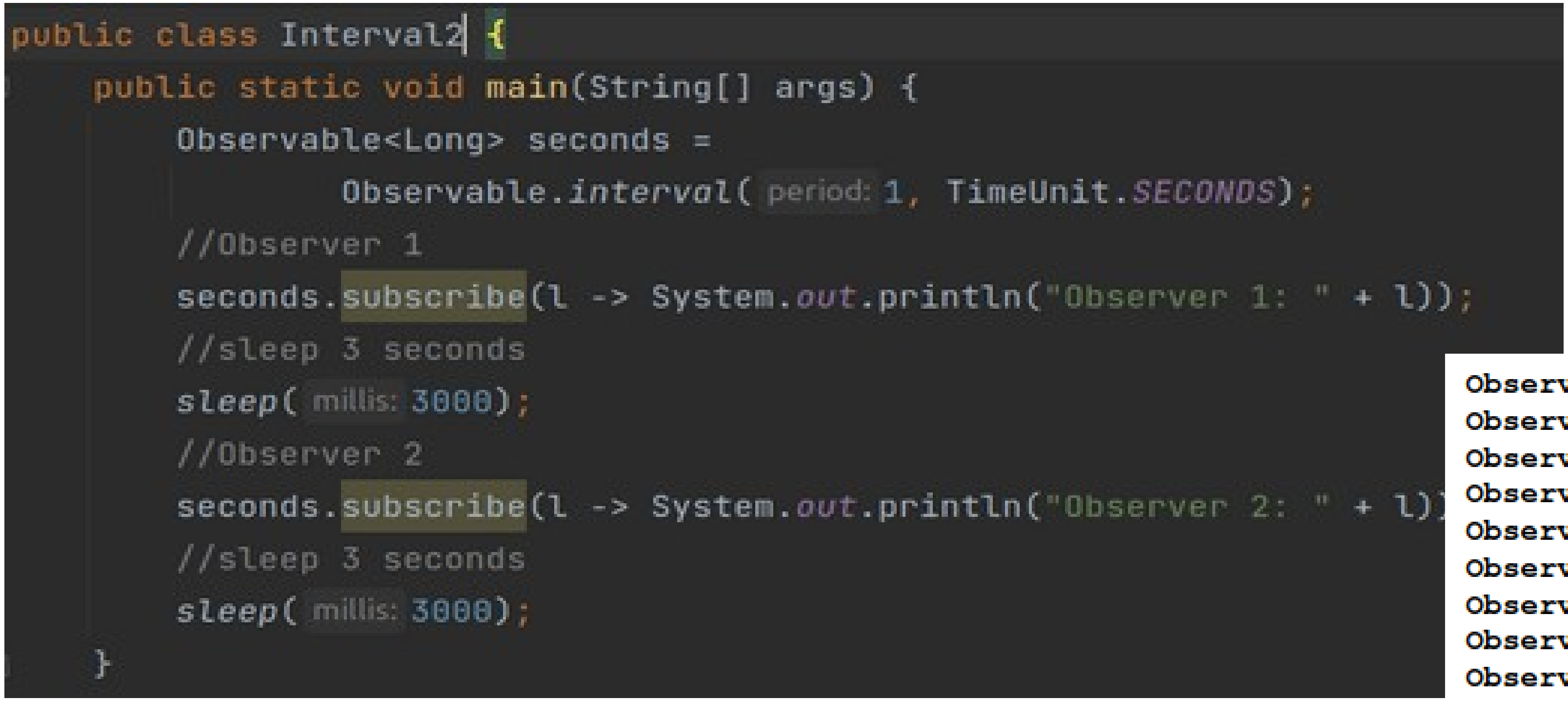
```
34 0 Mississippi
35 1 Mississippi
36 2 Mississippi
```

## Observables and Observer:Interval

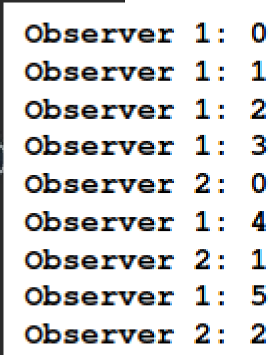
- Observable.interval() emits infinitely at the specified interval (which is 1 second in our example).
  - However, because it operates on a timer, it needs to run on a **separate thread**, which is the **computation scheduler** by default.
  - To keep our main() method from finishing and exiting the application before our Observable has a chance to finish emitting, we use the sleep()

# Observables and Observer:Interval:Hot or cold?

- Does Observable.interval() return a hot or a cold Observable?
  - Because it is event-driven (and infinite), you may be tempted to say it is hot. But put a second Observer on it, wait for 3 seconds, and then add another Observer. What happens?



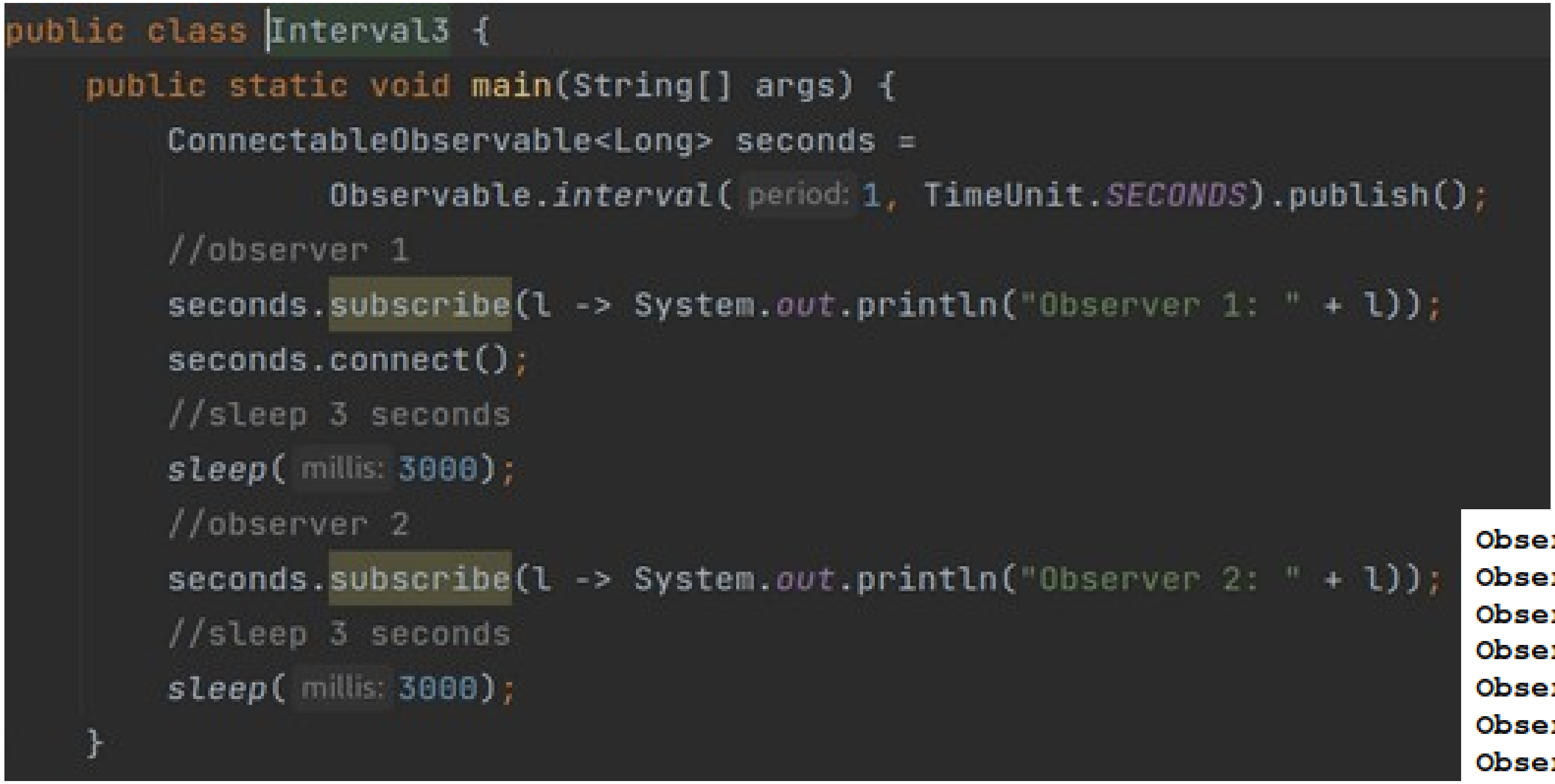
```
public class Interval2 {
    public static void main(String[] args) {
        Observable<Long> seconds =
                Observable.interval( period: 1, TimeUnit.SECONDS);
        //Observer 1
        seconds.subscribe(l -> System.out.println("Observer 1: " + l));
        //sleep 3 seconds
        sleep( millis: 3000);
        //Observer 2
        seconds.subscribe(l -> System.out.println("Observer 2: " + l))
        //sleep 3 seconds
        sleep( millis: 3000);
    }
```



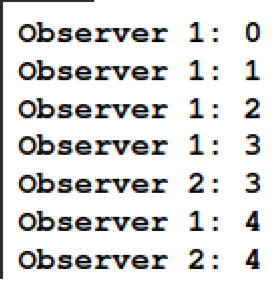
```
Observer 1: 0
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 0
Observer 1: 4
Observer 2: 1
Observer 1: 5
Observer 2: 2
```

## Observables and Observer:Interval:Hot or cold?

- To put all observers on the same timer with the same emission, you can use ConnectableObservable, which forces these emissions to become hot:



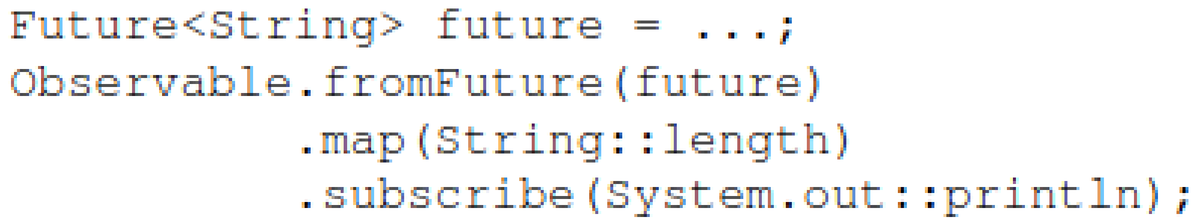
```java
public class Interval3 {
    public static void main(String[] args) {
        ConnectableObservable<Long> seconds =
                Observable.interval( period: 1, TimeUnit.SECONDS).publish();
        //observer 1
        seconds.subscribe(l -> System.out.println("Observer 1: " + l));
        seconds.connect();
        //sleep 3 seconds
        sleep( millis: 3000);
        //observer 2
        seconds.subscribe(l -> System.out.println("Observer 2: " + l));
        //sleep 3 seconds
        sleep( millis: 3000);
    }
```



```
Observer 1: 0
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 3
Observer 1: 4
Observer 2: 4
```

# Observables and Observer:future

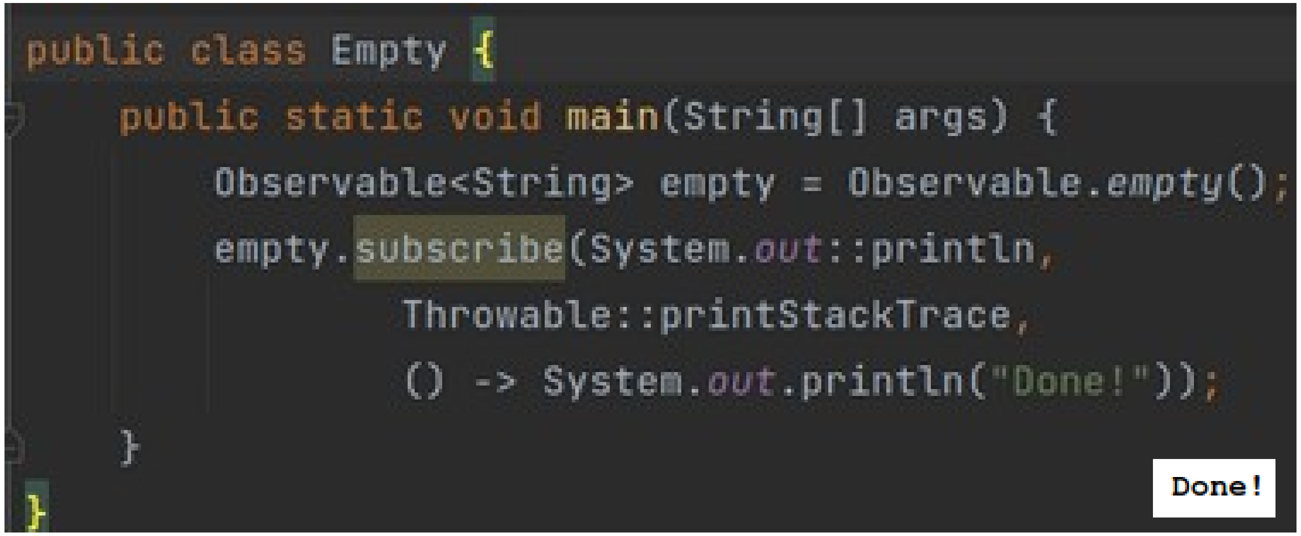
- Although this may not seem useful yet, it is sometimes helpful to create an Observable that emits nothing and calls onComplete():



```
Future<String> future = ...;
Observable.fromFuture(future)
          .map(String::length)
          .subscribe(System.out::println);
```

## Observables and Observer:empty

- Although this may not seem useful yet, it is sometimes helpful to create an Observable that emits nothing and calls onComplete():

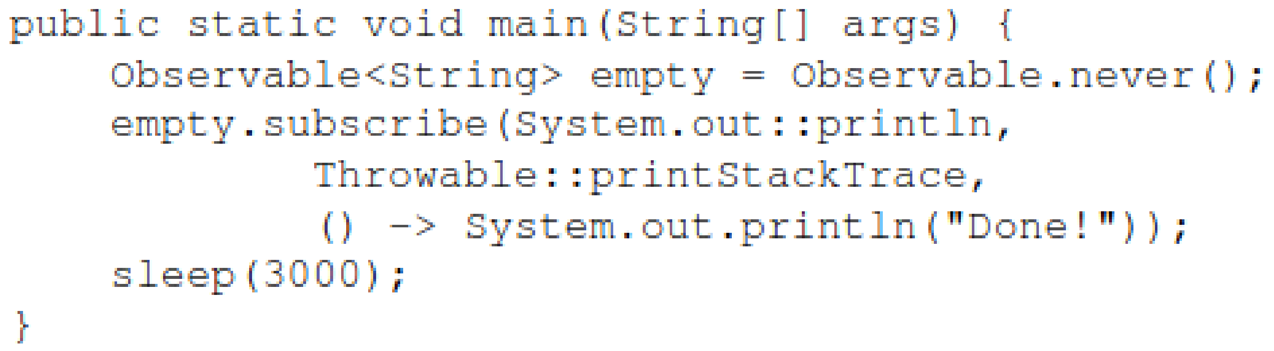


```
public class Empty {
    public static void main(String[] args) {
        Observable<String> empty = Observable.empty();
        empty.subscribe(System.out::println,
                Throwable::printStackTrace,
                () -> System.out.println("Done!"));
    }
}
```

```
Done!
```

## Observables and Observer:never

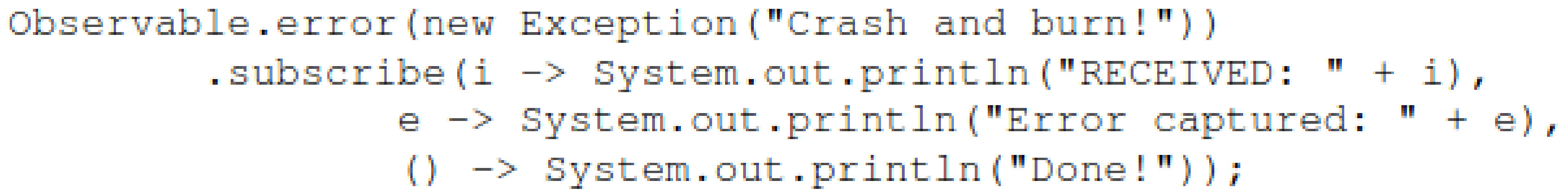
- A close cousin of Observable.empty() is Observable.never(). The only difference between them is that the never() method does not generate the onComplete event, thus leaving the observer waiting for an emission forever:



```
public static void main(String[] args) {
    Observable<String> empty = Observable.never();
    empty.subscribe(System.out::println,
            Throwable::printStackTrace,
            () -> System.out.println("Done!"));
    sleep(3000);
}
```

## Observables and Observer:error

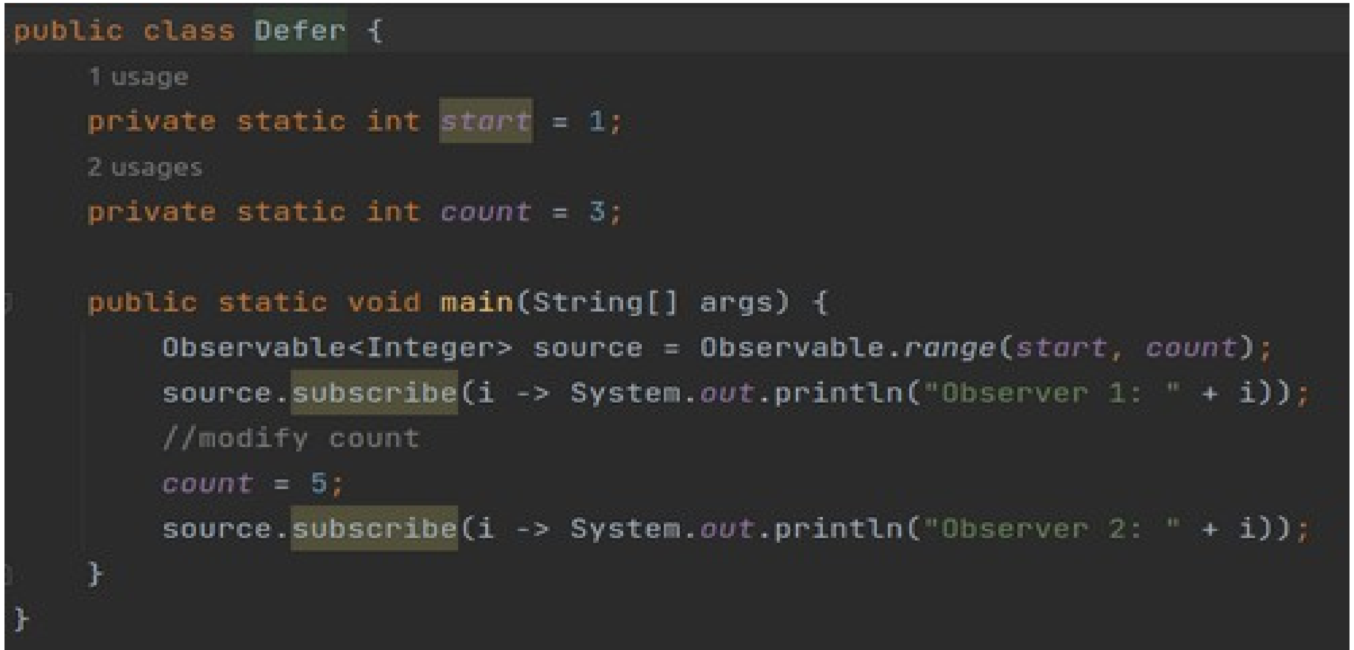
- This, too, is something you likely will use only with testing. It creates an Observable that immediately generates an onError event with the specified exception:



```
Observable.error(new Exception("Crash and burn!"))
        .subscribe(i -> System.out.println("RECEIVED: " + i),
                e -> System.out.println("Error captured: " + e),
                () -> System.out.println("Done!"));
```

## Observables and Observer:defer

- Observable.defer() is a powerful factory due to its ability to create a separate state for each Observer.
  - When using certain Observable factories, you may run into some nuances if your source is stateful and you want to create a separate state for each Observer.

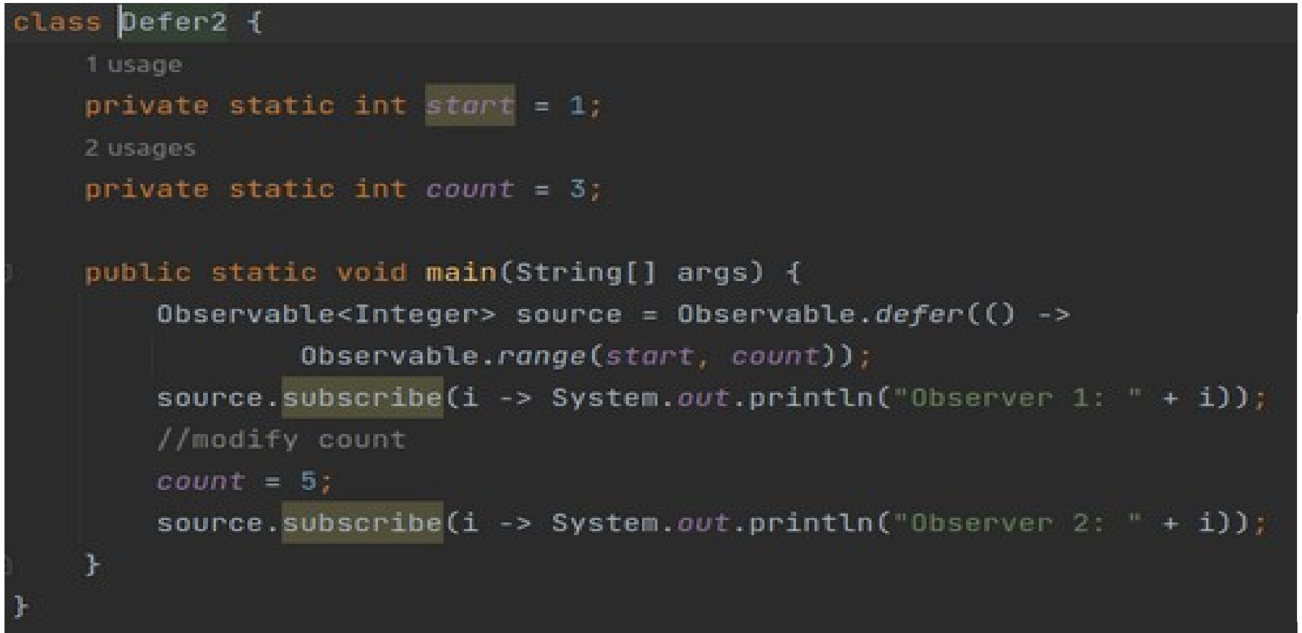


```java
public class Defer {
    1 usage
    private static int start = 1;
    2 usages
    private static int count = 3;

    public static void main(String[] args) {
        Observable<Integer> source = Observable.range(start, count);
        source.subscribe(i -> System.out.println("Observer 1: " + i));
        //modify count
        count = 5;
        source.subscribe(i -> System.out.println("Observer 2: " + i));
    }
}
```

```
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 1
Observer 2: 2
Observer 2: 3
```

# Observables and Observer:defer

- Observable.defer() is a powerful factory due to its ability to create a separate state for each Observer.
  - Observable sources not capturing state changes, you can create
  - a fresh Observable for each subscription. This can be achieved using **Observable.defer()**, which accepts a lambda expression.



```
class Defer2 {
    1 usage
    private static int start = 1;
    2 usages
    private static int count = 3;

    public static void main(String[] args) {
        Observable<Integer> source = Observable.defer(() ->
                Observable.range(start, count));
        source.subscribe(i -> System.out.println("Observer 1: " + i));
        //modify count
        count = 5;
        source.subscribe(i -> System.out.println("Observer 2: " + i));
    }
}
```

```
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 1
Observer 2: 2
Observer 2: 3
Observer 2: 4
Observer 2: 5
```

## Observables and Observer:fromCallable

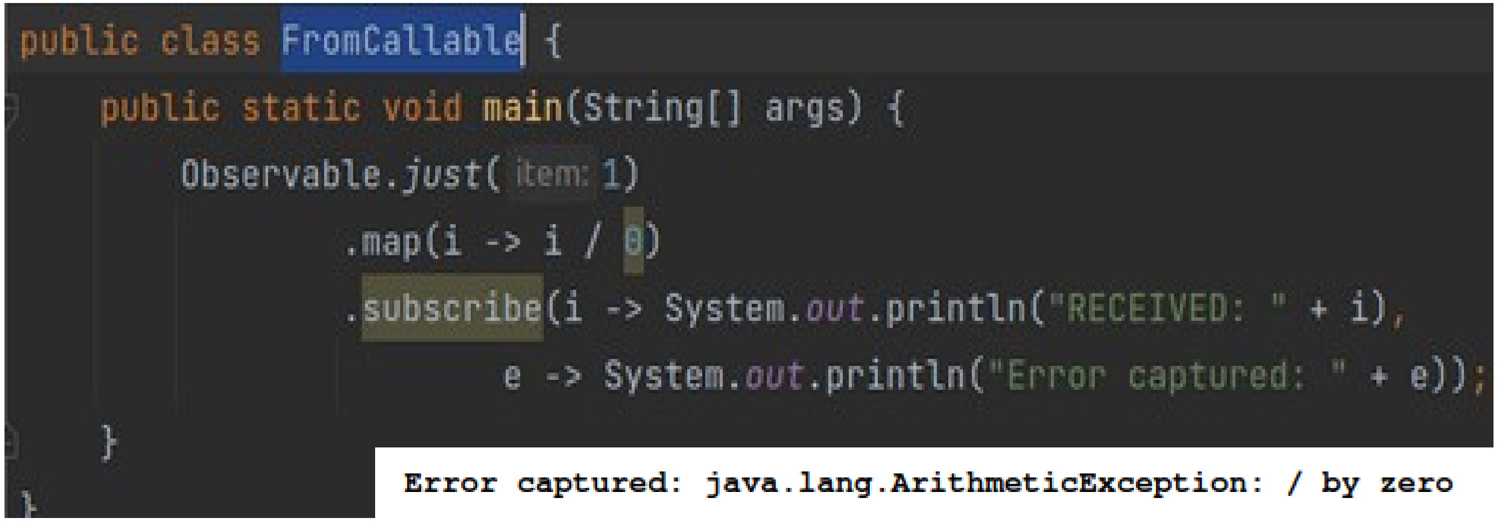
- If you need to perform a calculation or some other action and then emit the result, you can use Observable.just() (or Single.just() or Maybe.just()).
  - But sometimes, we want to do this in a lazy or deferred manner.
  - That is because the Observable was not even created, so we do not see the Error captured: message.

```
public static void main(String[] args) {
    Observable.just(1 / 0)
            .subscribe(i -> System.out.println("RECEIVED: " + i),
                    e -> System.out.println("Error captured: " + e));
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
<the stack trace follows>
```

## Observables and Observer:fromCallable

- If you need to perform a calculation or some other action and then emit the result, you can use Observable.just() (or Single.just() or Maybe.just()).
  - As you can see, the exception was captured by the Observer and successfully processed by the second lambda expression.

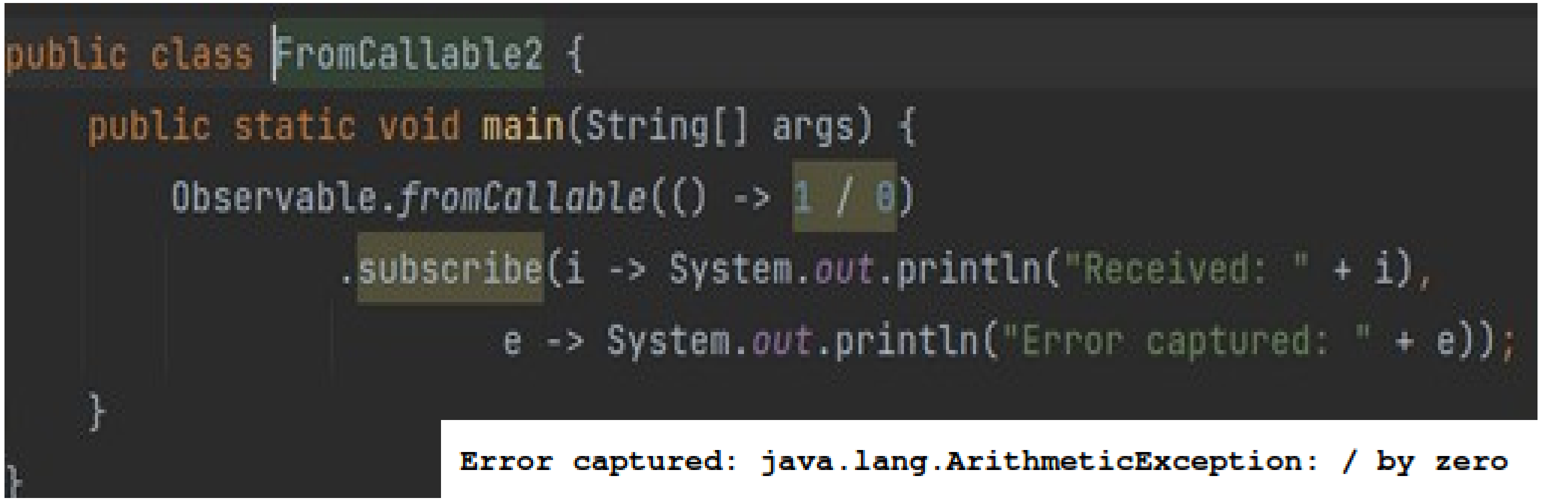


```
public class FromCallable {
    public static void main(String[] args) {
        Observable.just( item: 1)
                .map(i -> i / 0)
                .subscribe(i -> System.out.println("RECEIVED: " + i),
                        e -> System.out.println("Error captured: " + e));
    }
}
```

```
Error captured: java.lang.ArithmeticException: / by zero
```

## Observables and Observer:fromCallable

- If you want it to be emitted down the Observable chain along with an onError event, even if thrown during the emission initialization, use Observable.fromCallable() instead.
  - It accepts a functional interface, Supplier<T>.

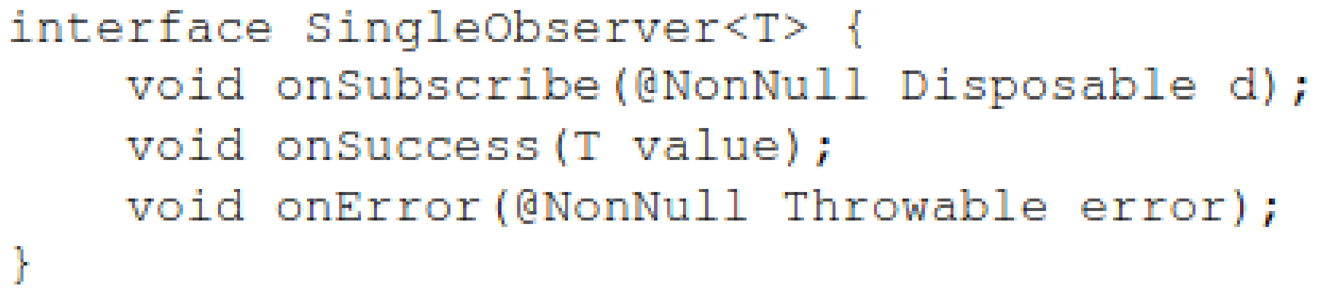


```java
public class FromCallable2 {
    public static void main(String[] args) {
        Observable.fromCallable(() -> 1 / 0)
                .subscribe(i -> System.out.println("Received: " + i),
                        e -> System.out.println("Error captured: " + e));
    }
}
```

```
Error captured: java.lang.ArithmeticException: / by zero
```

# Observables and Observer:Single

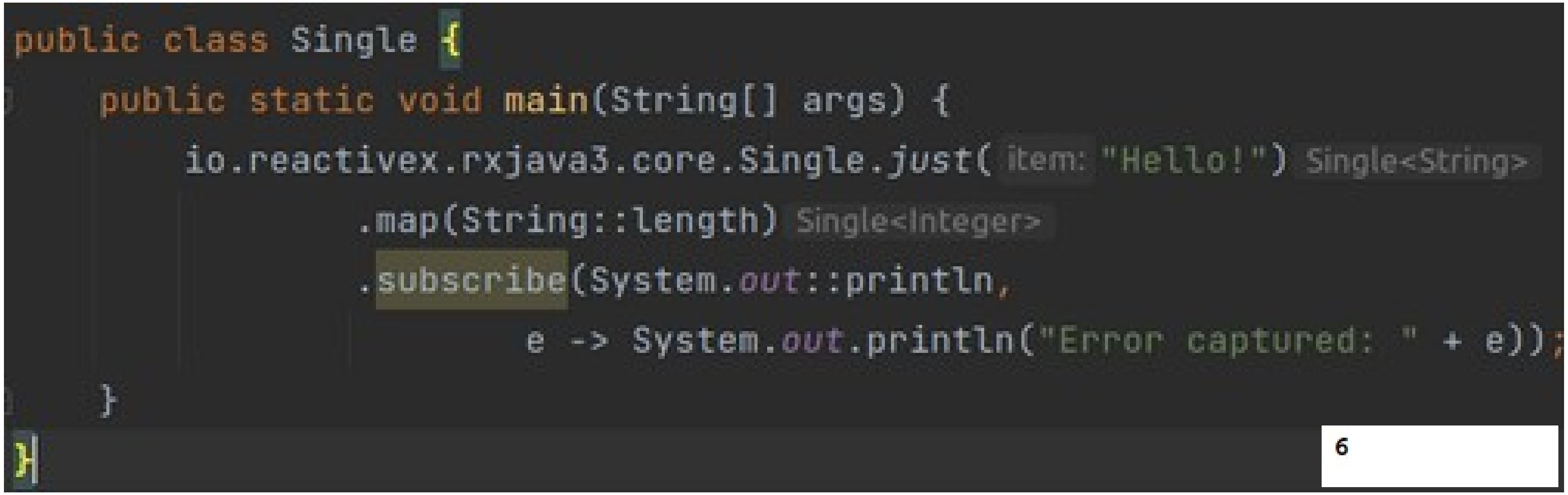
- The Single<T> class is essentially an Observable<T> that emits only one item and, as such, is limited only to operators that make sense for a single emission.
  - Similar to the Observable class (which implements ObservableSource), the Single class implements the SingleSource functional interface, which has only one method, void subscribe(SingleObserver observer).



```
interface SingleObserver<T> {
    void onSubscribe(@NonNull Disposable d);
    void onSuccess(T value);
    void onError(@NonNull Throwable error);
}
```

# Observables and Observer:Single

- The SingleObserver interface **does not have the onNext()** method and has the **onSuccess()** method instead of onComplete().
  - This makes sense because Single can emit one value at the most. onSuccess() essentially consolidates onNext() and onComplete() into a single event.

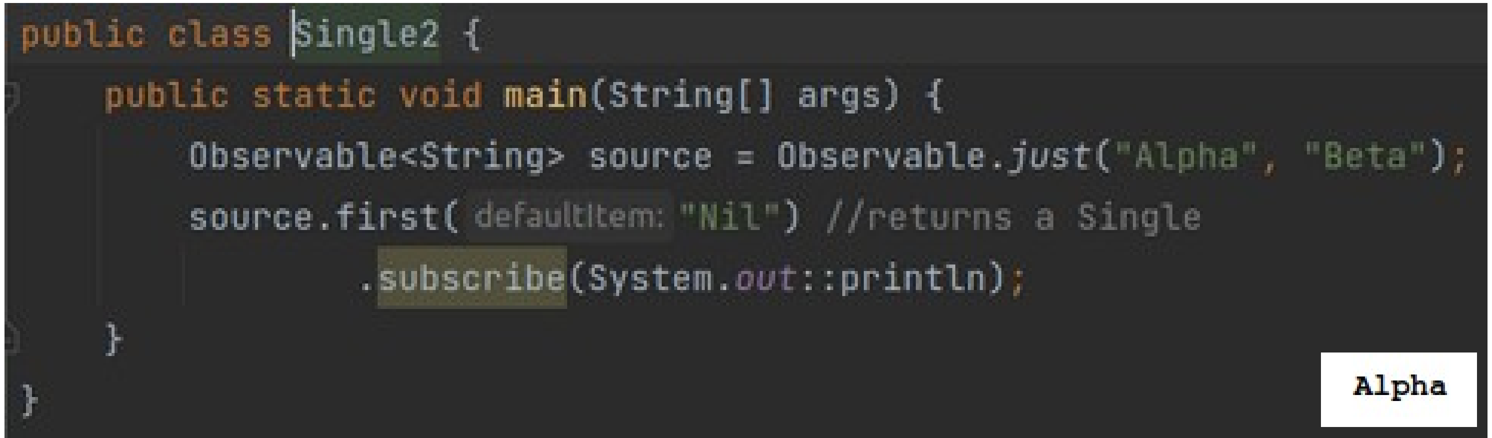


```
public class Single {
    public static void main(String[] args) {
        io.reactivex.rxjava3.core.Single.just( item: "Hello!") Single<String>
                .map(String::length) Single<Integer>
                .subscribe(System.out::println,
                        e -> System.out.println("Error captured: " + e));
    }
}
```

```
6
```

# Observables and Observer:Single

- There are operators on Single that turn it into an Observable, such as toObservable(). And, in the opposite direction, certain Observable operators return a Single
  - For instance, the first() operator will return a Single because that operator is logically concerned with a single item.
  - However, it accepts a default value as a parameter (which is specified as Nil in the following example) if the Observable comes out empty

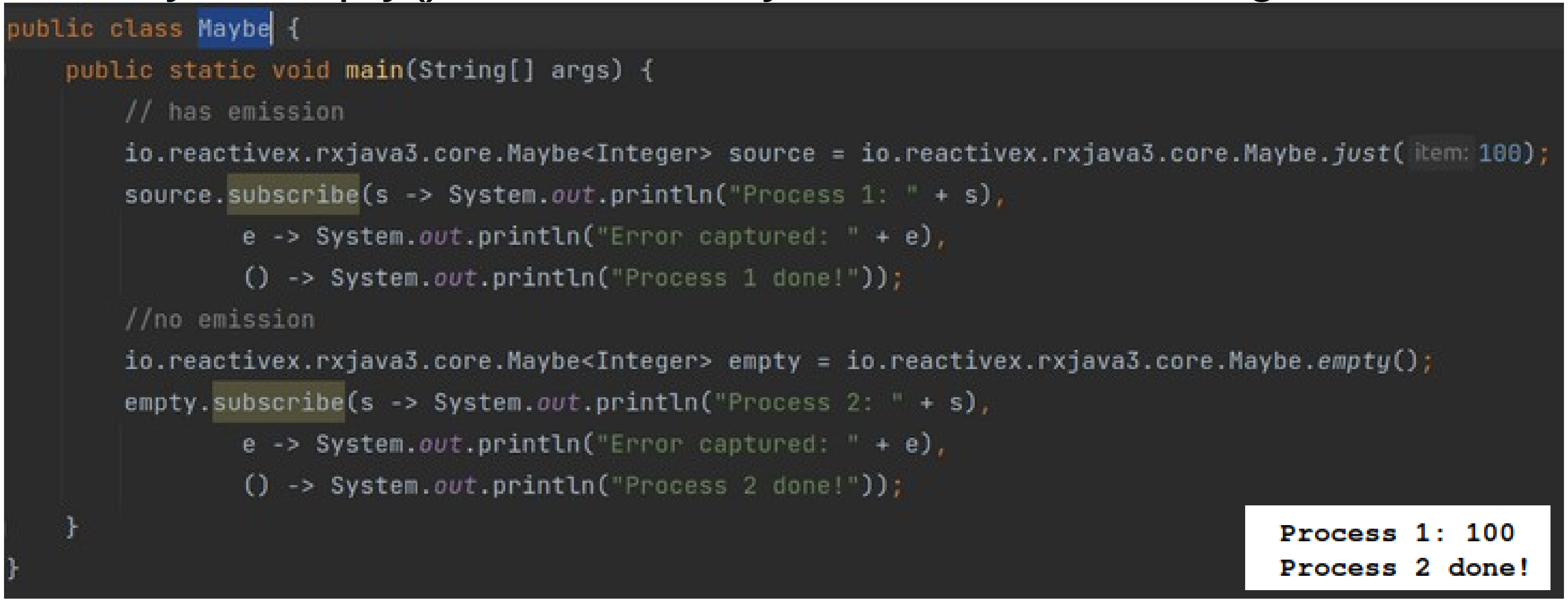


```
public class Single2 {
    public static void main(String[] args) {
        Observable<String> source = Observable.just("Alpha", "Beta");
        source.first( defaultItem: "Nil") //returns a Single
                .subscribe(System.out::println);
    }
}
```

```
Alpha
```

# Observables and Observer:Maybe

- A given Maybe<T> emits 0 or 1 items. It will pass the possible emission to onSuccess(), and in either case, it will call onComplete() when done. Maybe.just() can be used to create a Maybe emitting a single item.
  - Maybe.empty() creates a Maybe that emits nothing:

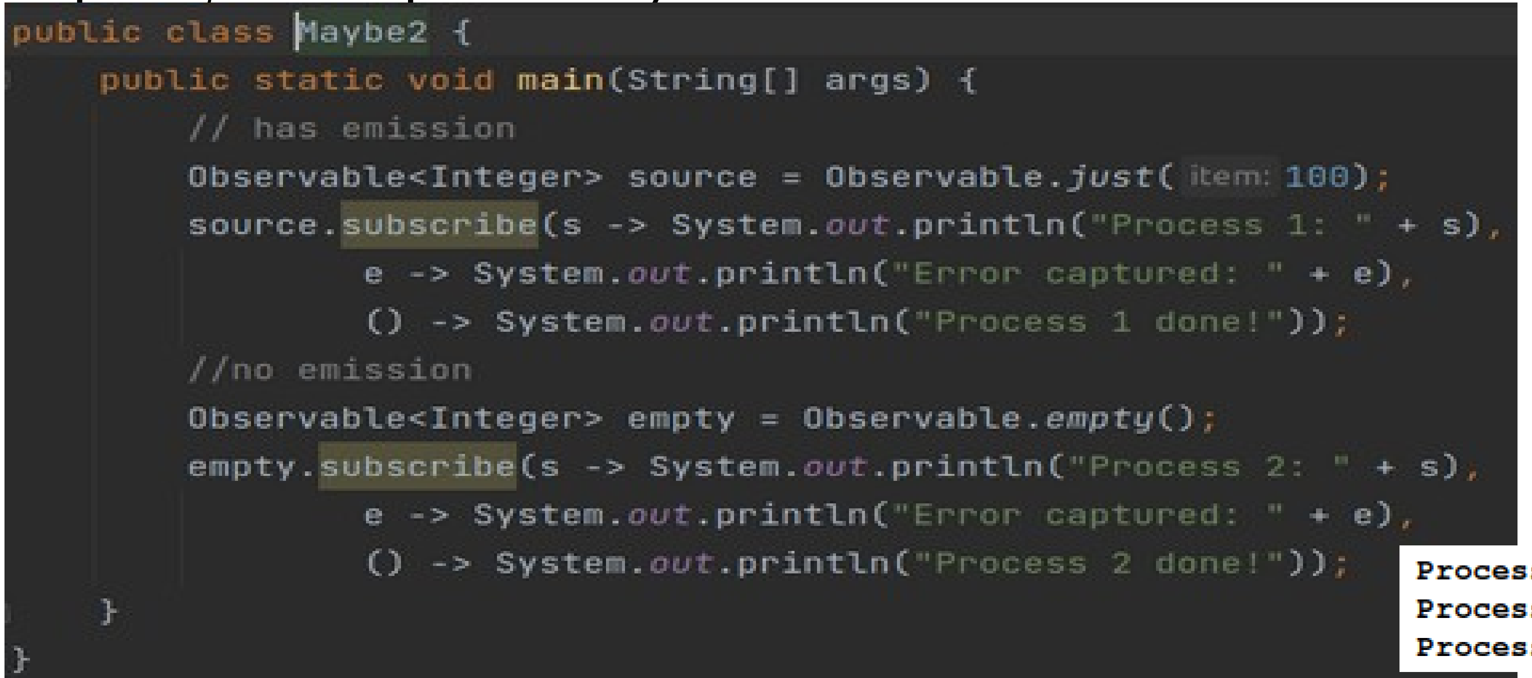


```
public class Maybe {
    public static void main(String[] args) {
        // has emission
        io.reactivex.rxjava3.core.Maybe<Integer> source = io.reactivex.rxjava3.core.Maybe.just( item: 100);
        source.subscribe(s -> System.out.println("Process 1: " + s),
                e -> System.out.println("Error captured: " + e),
                () -> System.out.println("Process 1 done!"));
        //no emission
        io.reactivex.rxjava3.core.Maybe<Integer> empty = io.reactivex.rxjava3.core.Maybe.empty();
        empty.subscribe(s -> System.out.println("Process 2: " + s),
                e -> System.out.println("Error captured: " + e),
                () -> System.out.println("Process 2 done!"));
    }
}
```

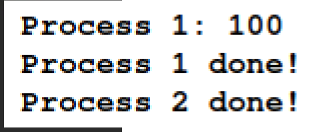
```
Process 1: 100
Process 2 done!
```

# Observables and Observer:Maybe

- The message Process 1 done! does not come up because there is no ambiguity: the Maybe observable cannot emit more than one item, so it is completed implicitly.
  - And MaybeObserver does not expect anything else. To prove this point, let's replace Maybe with Observable:



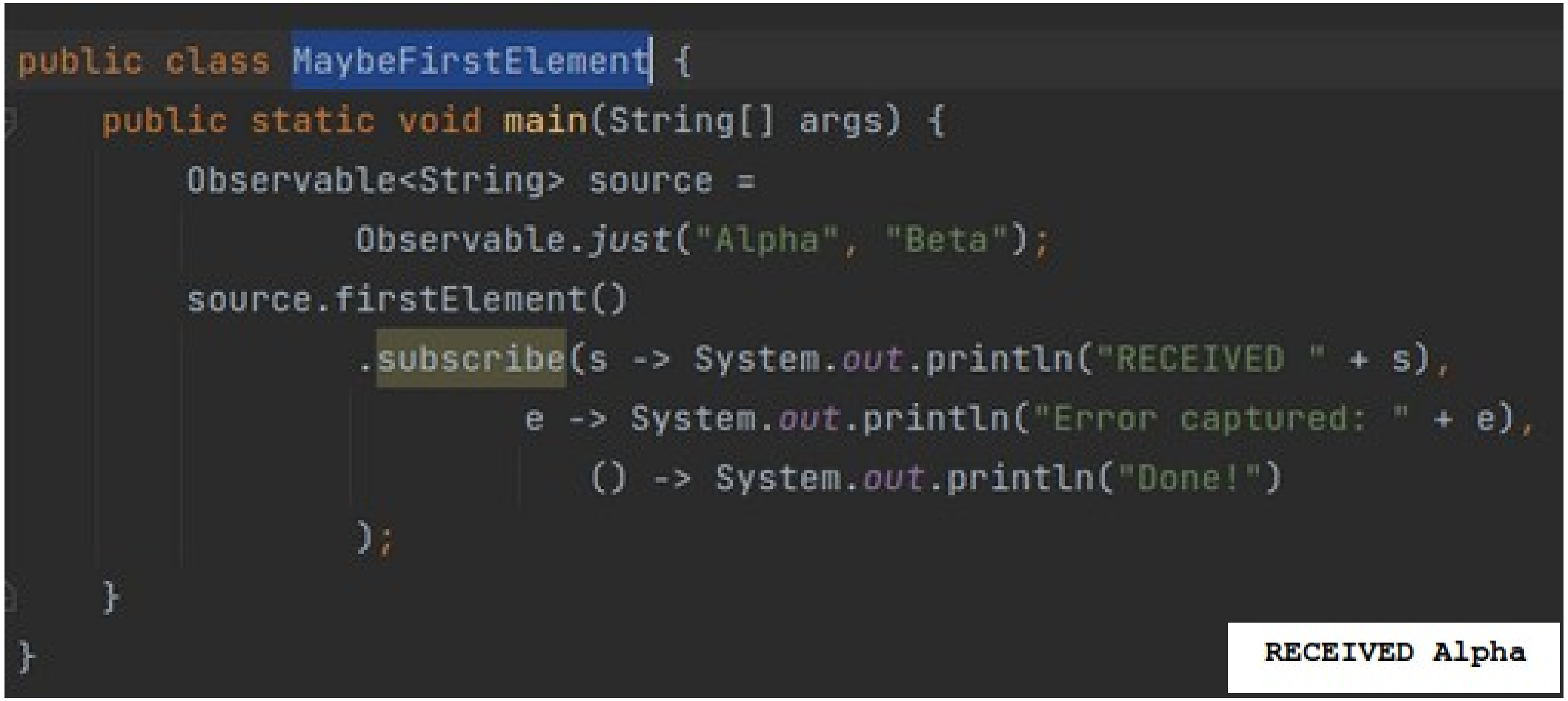
```
public class Maybe2 {
    public static void main(String[] args) {
        // has emission
        Observable<Integer> source = Observable.just( item: 100);
        source.subscribe(s -> System.out.println("Process 1: " + s),
                e -> System.out.println("Error captured: " + e),
                () -> System.out.println("Process 1 done!"));
        //no emission
        Observable<Integer> empty = Observable.empty();
        empty.subscribe(s -> System.out.println("Process 2: " + s),
                e -> System.out.println("Error captured: " + e),
                () -> System.out.println("Process 2 done!"));
    }
}
```



```
Process 1: 100
Process 1 done!
Process 2 done!
```

## Observables and Observer:Maybe

- Certain Observable operators that we will discuss later yield a Maybe. One example is the firstElement() operator, which is similar to first(), but returns an **empty result if no elements** are emitted:

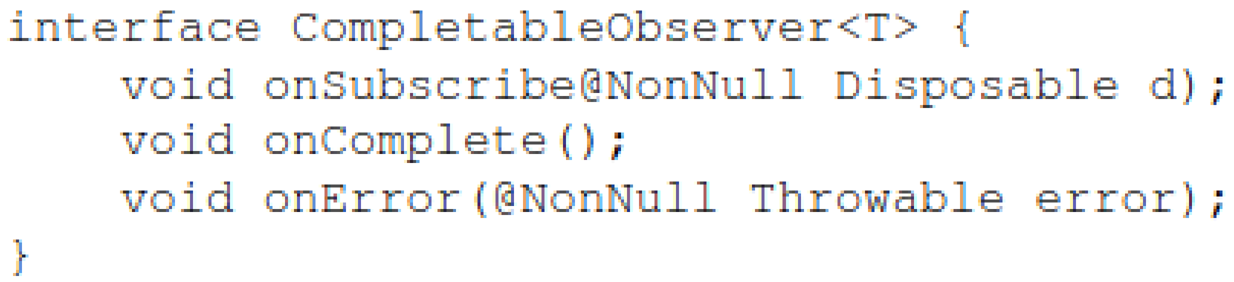


```
public class MaybeFirstElement {
    public static void main(String[] args) {
        Observable<String> source =
                Observable.just("Alpha", "Beta");
        source.firstElement()
                .subscribe(s -> System.out.println("RECEIVED " + s),
                        e -> System.out.println("Error captured: " + e),
                            () -> System.out.println("Done!")
                );
    }
}
```

```
RECEIVED Alpha
```

## Observables and Observer:Completable

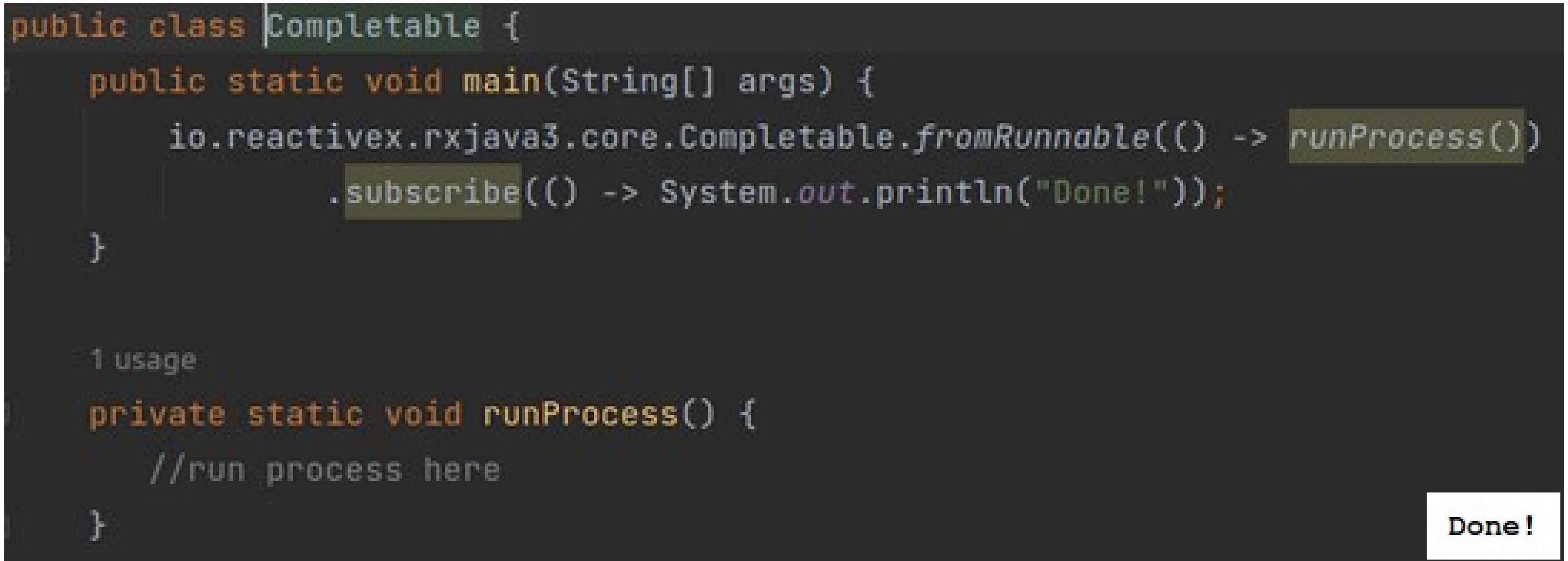
- Completable is simply concerned with an action being executed, but it does not receive any emissions. Logically, it does not have onNext() or onSuccess() to receive emissions, but it does have onError() and onComplete():



```
interface CompletableObserver<T> {
    void onSubscribe@NonNull Disposable d);
    void onComplete();
    void onError(@NonNull Throwable error);
}
```

# Observables and Observer:Completable

- Completable is something you likely will not use often. You can construct one quickly by calling Completable.complete() or Completable.fromRunnable().
  - The former immediately calls onComplete() without doing anything, while fromRunnable() executes the specified action before calling onComplete():



```
public class Completable {
    public static void main(String[] args) {
        io.reactivex.rxjava3.core.Completable.fromRunnable(() -> runProcess())
                .subscribe(() -> System.out.println("Done!"));
    }

    1 usage
    private static void runProcess() {
        //run process here
    }
```

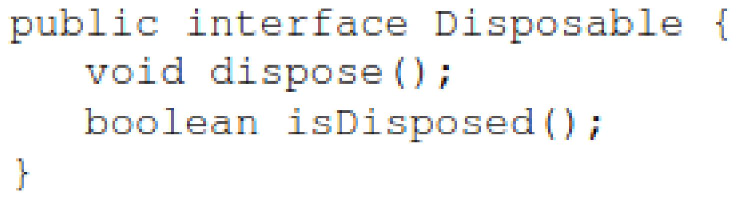
```
Done!
```

## Observables and Observer:Disposable

- When you call subscribe() to an Observable to receive emissions, a stream is created to process those emissions through the Observable chain.
  - Of course, this uses resources. When we are done, we want to dispose of these resources so that they can be garbage-collected.
  - The finite Observable that calls **onComplete()** will **typically dispose of itself safely** when all items are emitted.

## Observables and Observer:Disposable

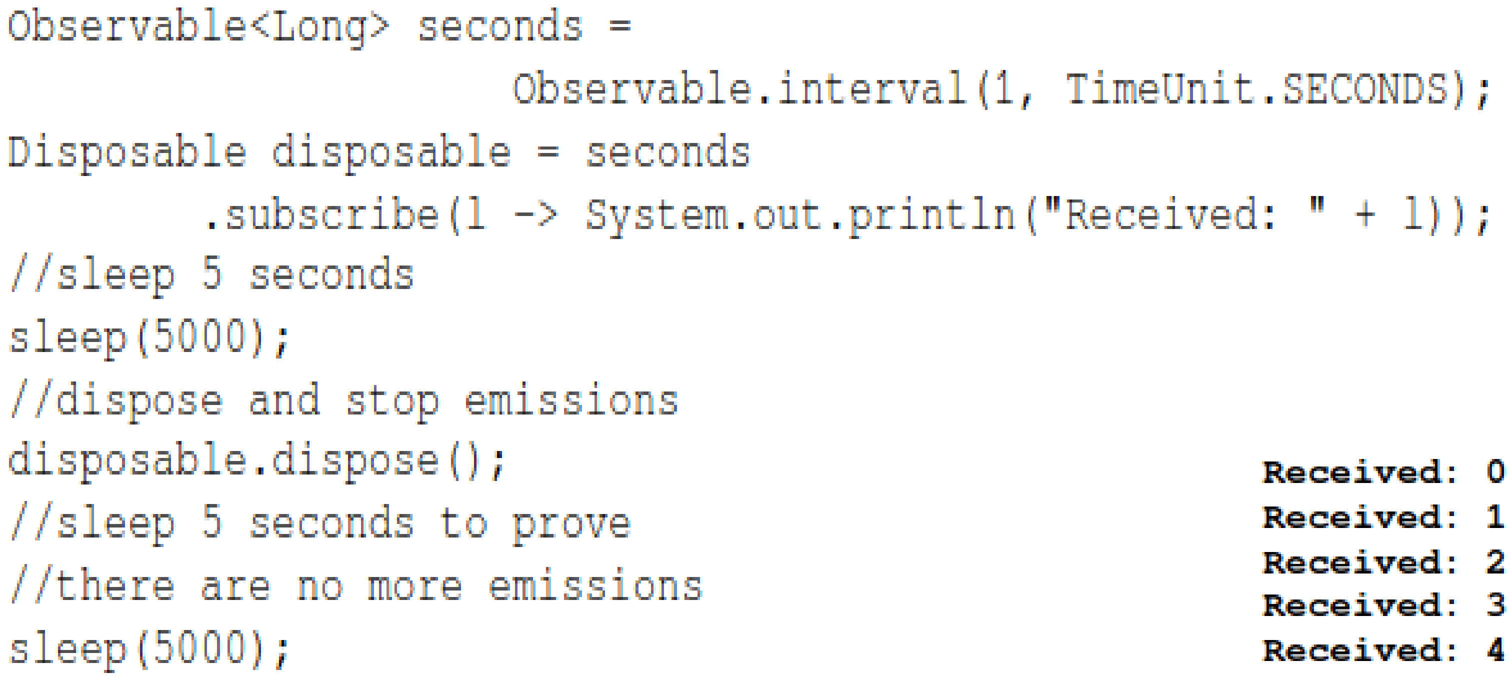
- When you call subscribe() to an Observable to receive emissions, a stream is created to process those emissions through the Observable chain.
  - But if you are working with an **infinite or long-running** Observable, you likely will run into situations where you want to **explicitly stop the emissions and dispose of everything associated with that subscription**.
  - Garbage collector **cannot collect active subscriptions** that you no longer need, and explicit disposal is necessary in order to prevent memory leaks.



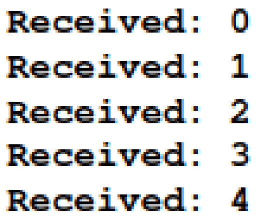
```
public interface Disposable {
   void dispose();
   boolean isDisposed();
}
```

## Observables and Observer:Disposable

- The subscribe() methods that **accept lambda expressions (not Observer)** return a **Disposable**.
  - You can use this object to stop emissions at any time by **calling its dispose()** method.



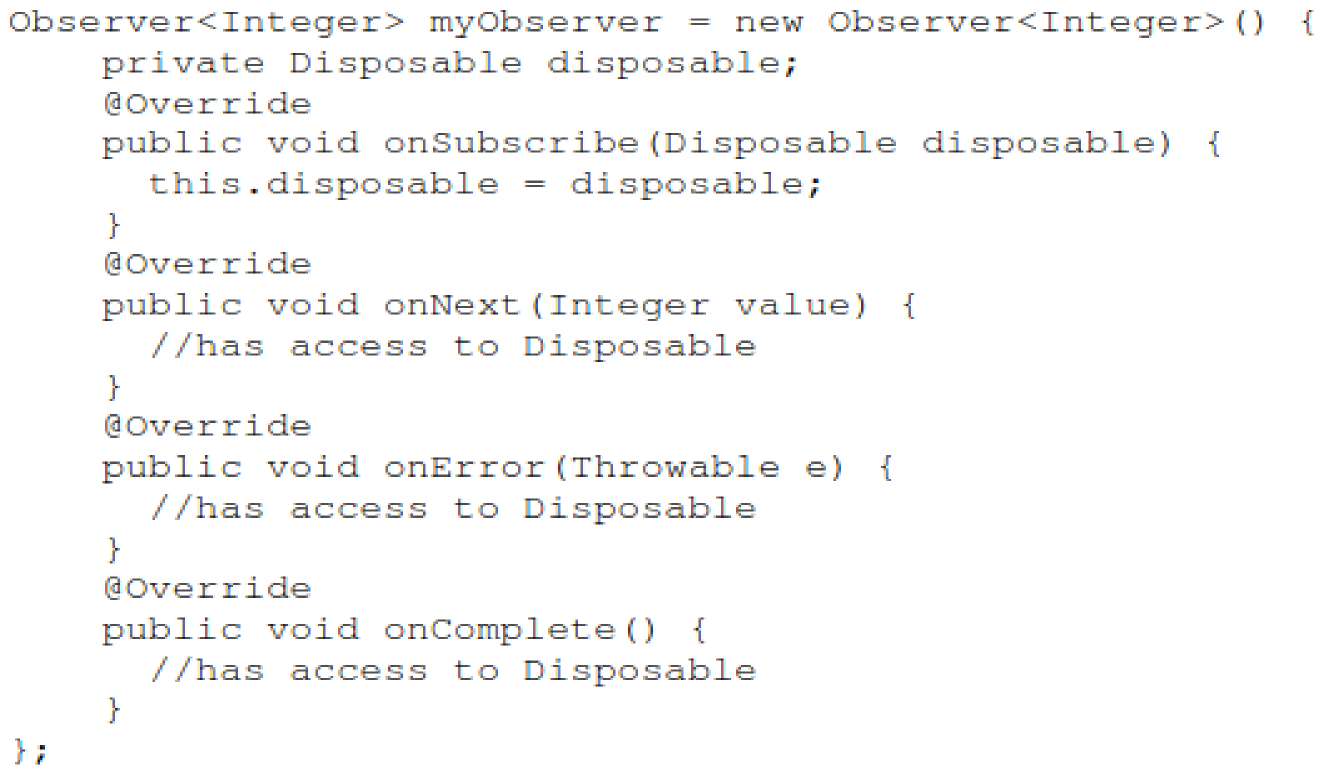
```
Observable<Long> seconds =
                     Observable.interval(1, TimeUnit.SECONDS);
Disposable disposable = seconds
        .subscribe(l -> System.out.println("Received: " + l));
//sleep 5 seconds
sleep(5000);
//dispose and stop emissions
disposable.dispose();
//sleep 5 seconds to prove
//there are no more emissions
sleep(5000);
```



```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
```

## Observables and Observer:Disposable

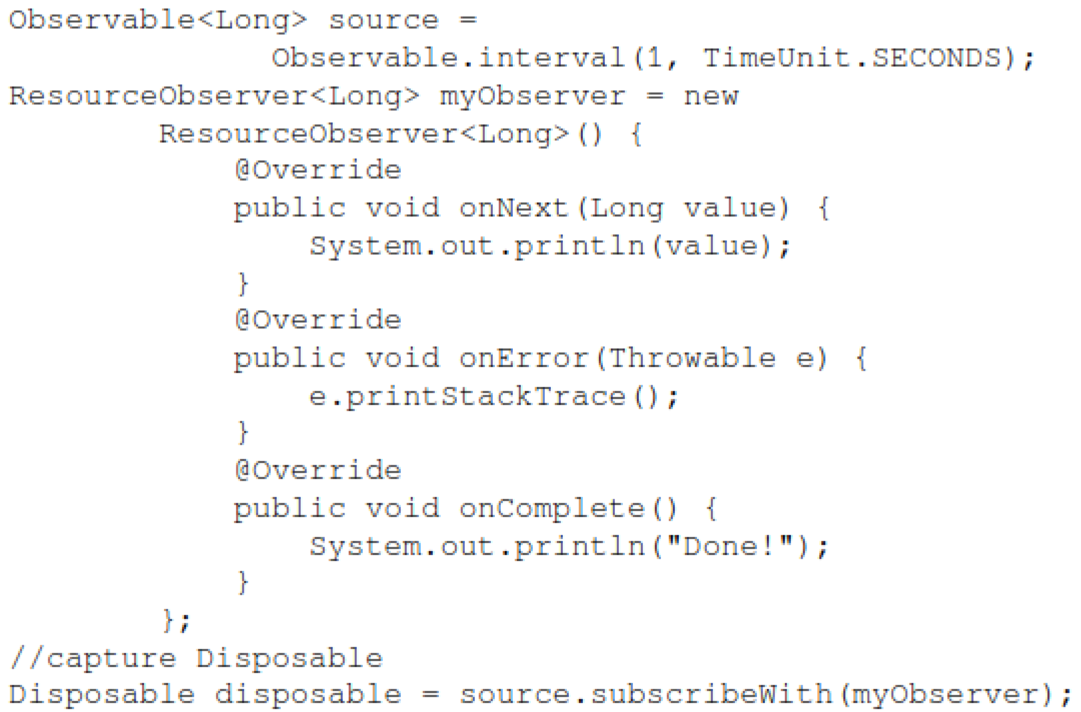
- The subscribe() method that accepts Observer returns void (not a Disposable) since it is assumed that the Observer will handle everything.



```
Observer<Integer> myObserver = new Observer<Integer>() {
    private Disposable disposable;
    @Override
    public void onSubscribe(Disposable disposable) {
      this.disposable = disposable;
    }
    @Override
    public void onNext(Integer value) {
      //has access to Disposable
    }
    @Override
    public void onError(Throwable e) {
      //has access to Disposable
    }
    @Override
    public void onComplete() {
      //has access to Disposable
    }
};
```

# Observables and Observer:Disposable

- Pass the ResourceObserver object to subscribeWith() instead of subscribe(), and you will get the default Disposable returned:
  - This is a compromise between the built-in observer or your observer handling it.



```
Observable<Long> source =
              Observable.interval(1, TimeUnit.SECONDS);
ResourceObserver<Long> myObserver = new
        ResourceObserver<Long>() {
            @Override
            public void onNext(Long value) {
                System.out.println(value);
            }
            @Override
            public void onError(Throwable e) {
                e.printStackTrace();
            }
            @Override
            public void onComplete() {
                System.out.println("Done!");
            }
        };
//capture Disposable
Disposable disposable = source.subscribeWith(myObserver);
```

## Observables and Observer:Disposable:Composite

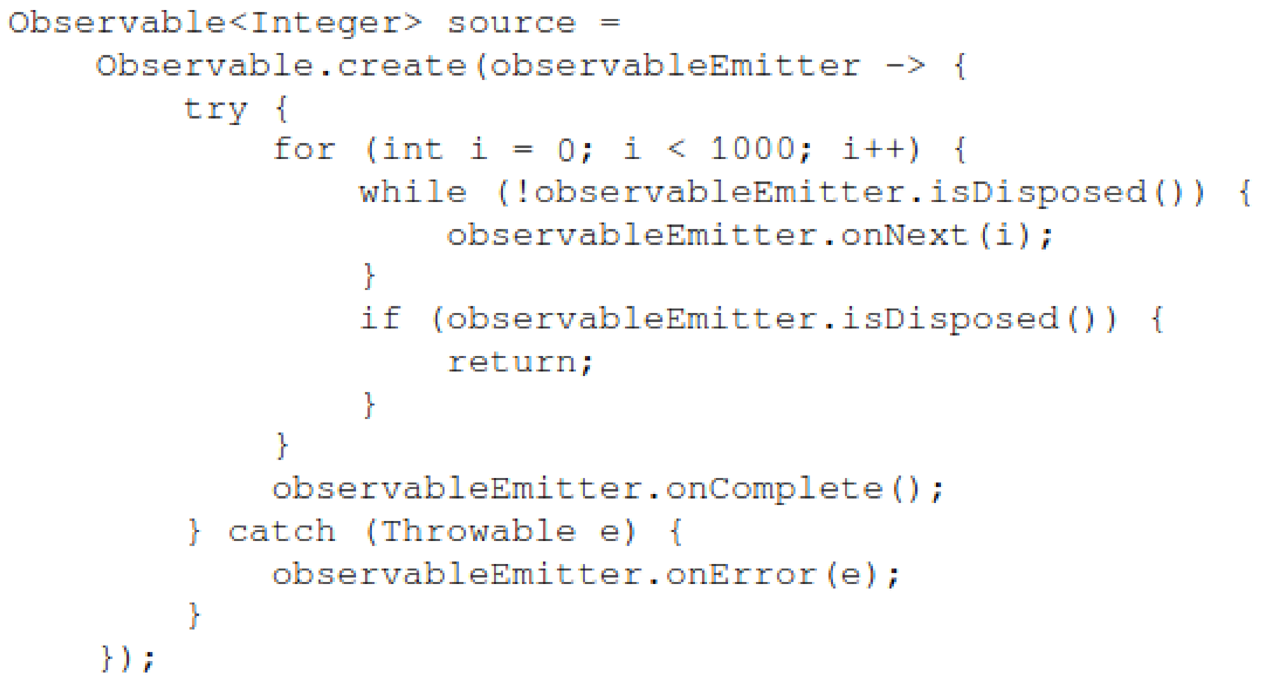
- This implements Disposable, but internally holds a collection of Disposable objects, which you can add to and then dispose of all at once:

```
Observable<Long> seconds =
                     Observable.interval(1, TimeUnit.SECONDS);
//subscribe and capture disposables
Disposable disposable1 = seconds
      .subscribe(l -> System.out.println("Observer 1: " + l));
Disposable disposable2 = seconds
      .subscribe(l -> System.out.println("Observer 2: " + l));
//put both disposables into CompositeDisposable
disposables.addAll(disposable1, disposable2);
//sleep 5 seconds
sleep(5000);
//dispose all disposables
disposables.dispose();
//sleep 5 seconds to prove
//there are no more emissions
sleep(5000);
```

```
Observer 1: 0
Observer 2: 0
Observer 1: 1
Observer 2: 1
Observer 1: 2
Observer 2: 2
Observer 1: 3
Observer 2: 3
Observer 1: 4
Observer 2: 4
```

## Observables and Observer:Disposable:Create

- If your Observable.create() is returning a long-running or infinite Observable, you should ideally check the isDisposed() method of ObservableEmitter regularly, to see whether you should keep sending emissions.
  - This prevents unnecessary work from being done if the subscription is no longer active



```
Observable<Integer> source =
    Observable.create(observableEmitter -> {
        try {
            for (int i = 0; i < 1000; i++) {
                while (!observableEmitter.isDisposed()) {
                    observableEmitter.onNext(i);
                }
                if (observableEmitter.isDisposed()) {
                    return;
                }
            }
            observableEmitter.onComplete();
        } catch (Throwable e) {
            observableEmitter.onError(e);
        }
    });
```

## Observables and Observer:Disposable:Create

- If your Observable.create() is returning a long-running or infinite Observable, you should ideally check the isDisposed() method of ObservableEmitter regularly, to see whether you should keep sending emissions.
  - This prevents unnecessary work from being done if the subscription is no longer active

```
Observable<Integer> source =
    Observable.create(observableEmitter -> {
        try {
            for (int i = 0; i < 1000; i++) {
                while (!observableEmitter.isDisposed()) {
                    observableEmitter.onNext(i);
                }
                if (observableEmitter.isDisposed()) {
                    return;
                }
            }
            observableEmitter.onComplete();
        } catch (Throwable e) {
            observableEmitter.onError(e);
        }
    });
```

# Agenda

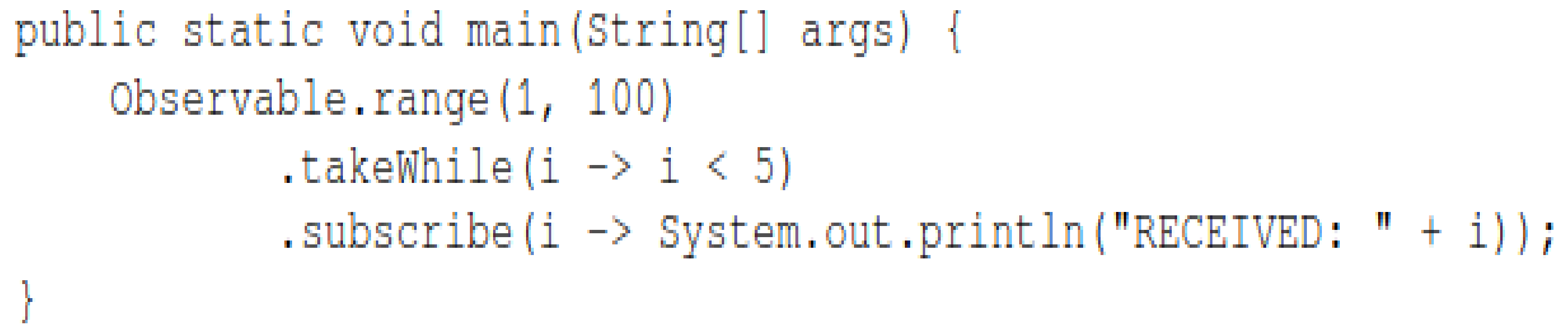
- Introduction
- Observables and Observer
- **Operators**
- Combining Observables
- Multicasting, Replaying and Caching
- Concurrency
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

## Operators:Conditional

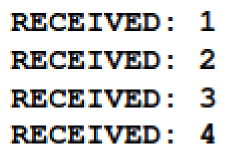
- Conditional operators emit or transform Observable conditionally.
  - This allows a control flow to be organized and the path of execution to be determined, which is especially important for adding decision-making ability to your program.

## Operators:Conditional

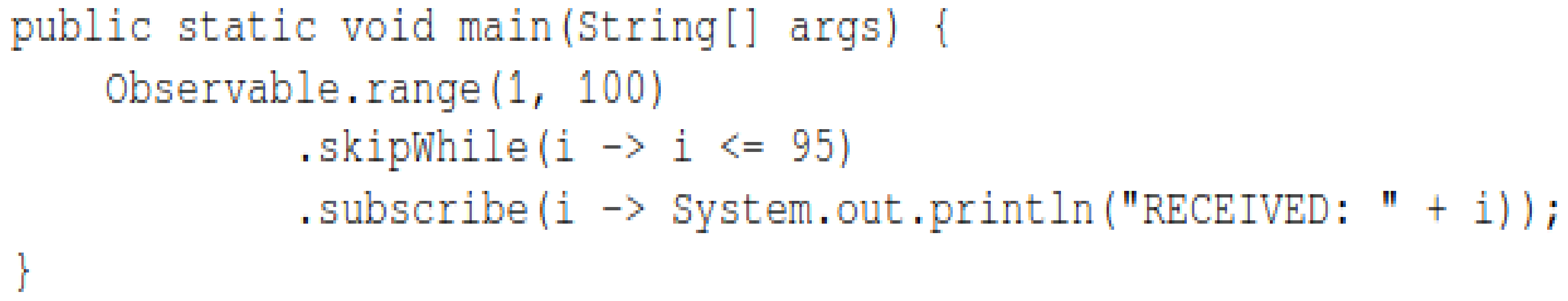
- **takeWhile() and skipWhile()**



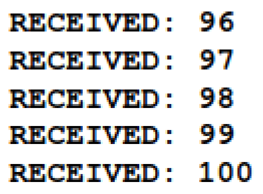
```
public static void main(String[] args) {
    Observable.range(1, 100)
            .takeWhile(i -> i < 5)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```



```
RECEIVED: 1
RECEIVED: 2
RECEIVED: 3
RECEIVED: 4
```



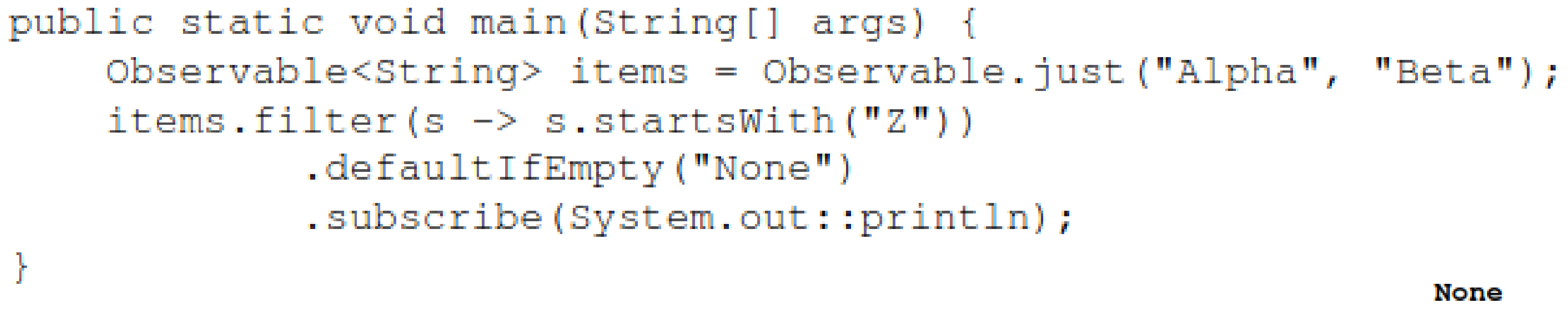
```
public static void main(String[] args) {
    Observable.range(1, 100)
            .skipWhile(i -> i <= 95)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```



```
RECEIVED: 96
RECEIVED: 97
RECEIVED: 98
RECEIVED: 99
RECEIVED: 100
```

## Operators:Conditional

- **defaultIfEmpty**

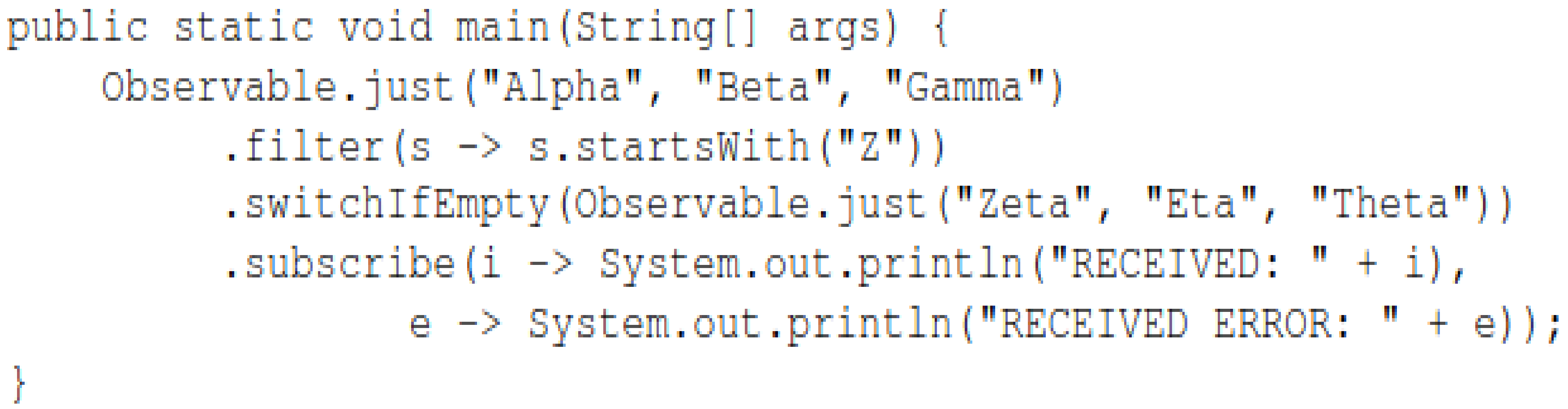


```java
public static void main(String[] args) {
    Observable<String> items = Observable.just("Alpha", "Beta");
    items.filter(s -> s.startsWith("Z"))
            .defaultIfEmpty("None")
            .subscribe(System.out::println);
}
```

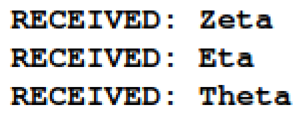
```
None
```

## Operators:Conditional

- **switchIfEmpty**



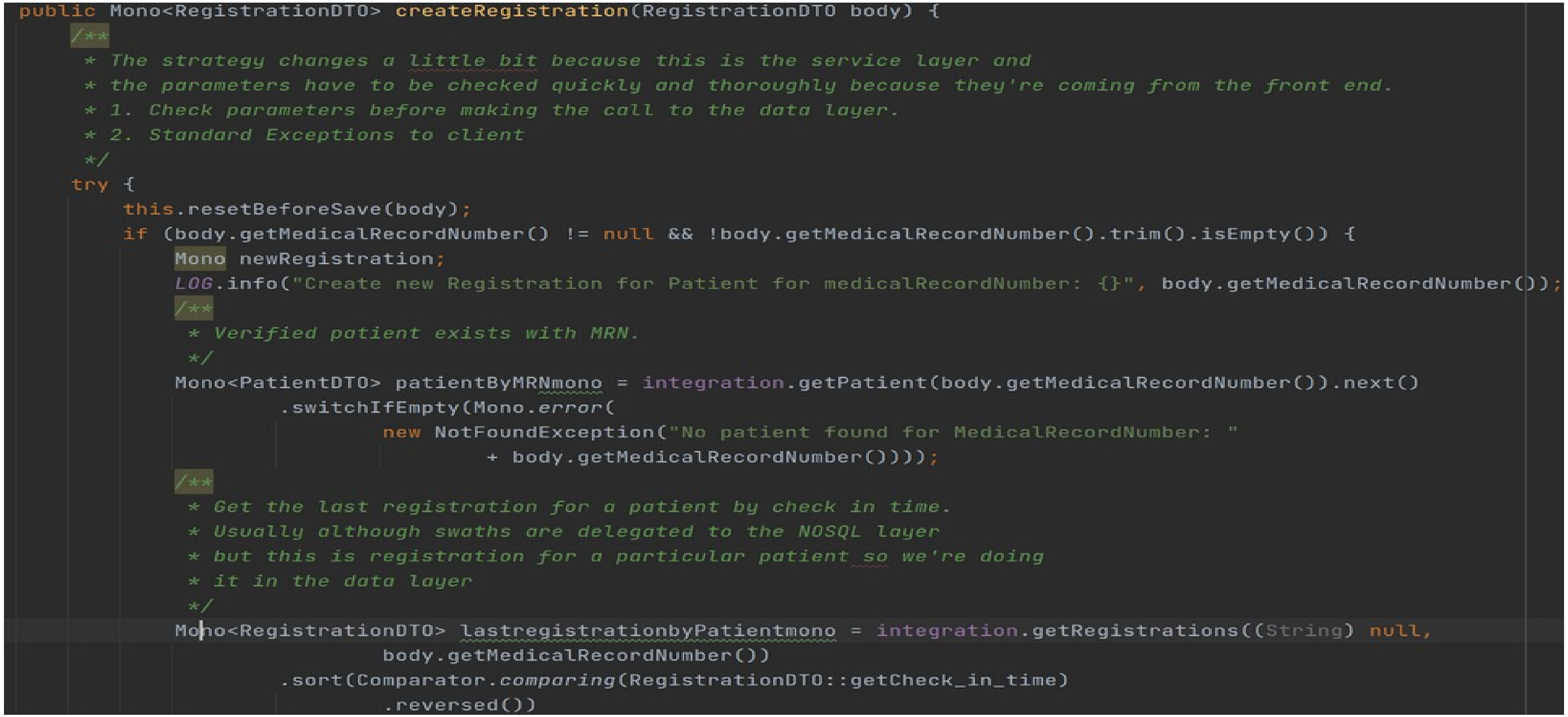
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
          .filter(s -> s.startsWith("Z"))
          .switchIfEmpty(Observable.just("Zeta", "Eta", "Theta"))
          .subscribe(i -> System.out.println("RECEIVED: " + i),
                  e -> System.out.println("RECEIVED ERROR: " + e));
}
```



```
RECEIVED: Zeta
RECEIVED: Eta
RECEIVED: Theta
```

## Operators:Conditional

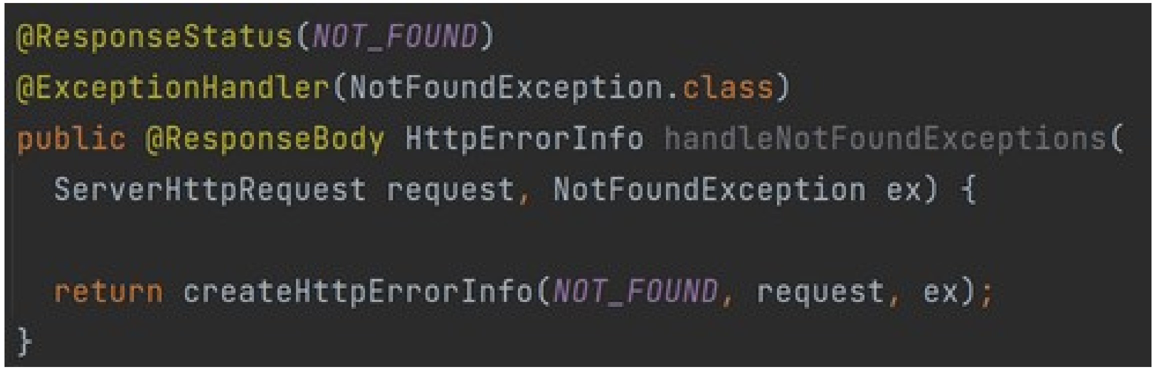
- **SwitchIfEmpty**
  - it is also used to throw exceptions and prevent further processing and converted to standard exceptions especially in case of http



```
public Mono<RegistrationDTO> createRegistration(RegistrationDTO body) {
    /**
     * The strategy changes a little bit because this is the service layer and
     * the parameters have to be checked quickly and thoroughly because they're coming from the front end.
     * 1. Check parameters before making the call to the data layer.
     * 2. Standard Exceptions to client
     */
    try {
        this.resetBeforeSave(body);
        if (body.getMedicalRecordNumber() != null && !body.getMedicalRecordNumber().trim().isEmpty()) {
            Mono newRegistration;
            LOG.info("Create new Registration for Patient for medicalRecordNumber: {}", body.getMedicalRecordNumber());
            /**
             * Verified patient exists with MRN.
             */
            Mono<PatientDTO> patientByMRNmono = integration.getPatient(body.getMedicalRecordNumber()).next()
                    .switchIfEmpty(Mono.error(
                            new NotFoundException("No patient found for MedicalRecordNumber: "
                                    + body.getMedicalRecordNumber())));
            /**
             * Get the last registration for a patient by check in time.
             * Usually although swaths are delegated to the NOSQL layer
             * but this is registration for a particular patient so we're doing
             * it in the data layer
             */
            Mono<RegistrationDTO> lastregistrationbyPatientmono = integration.getRegistrations((String) null,
                            body.getMedicalRecordNumber())
                    .sort(Comparator.comparing(RegistrationDTO::getCheck_in_time)
                            .reversed())
```

## Operators:Conditional

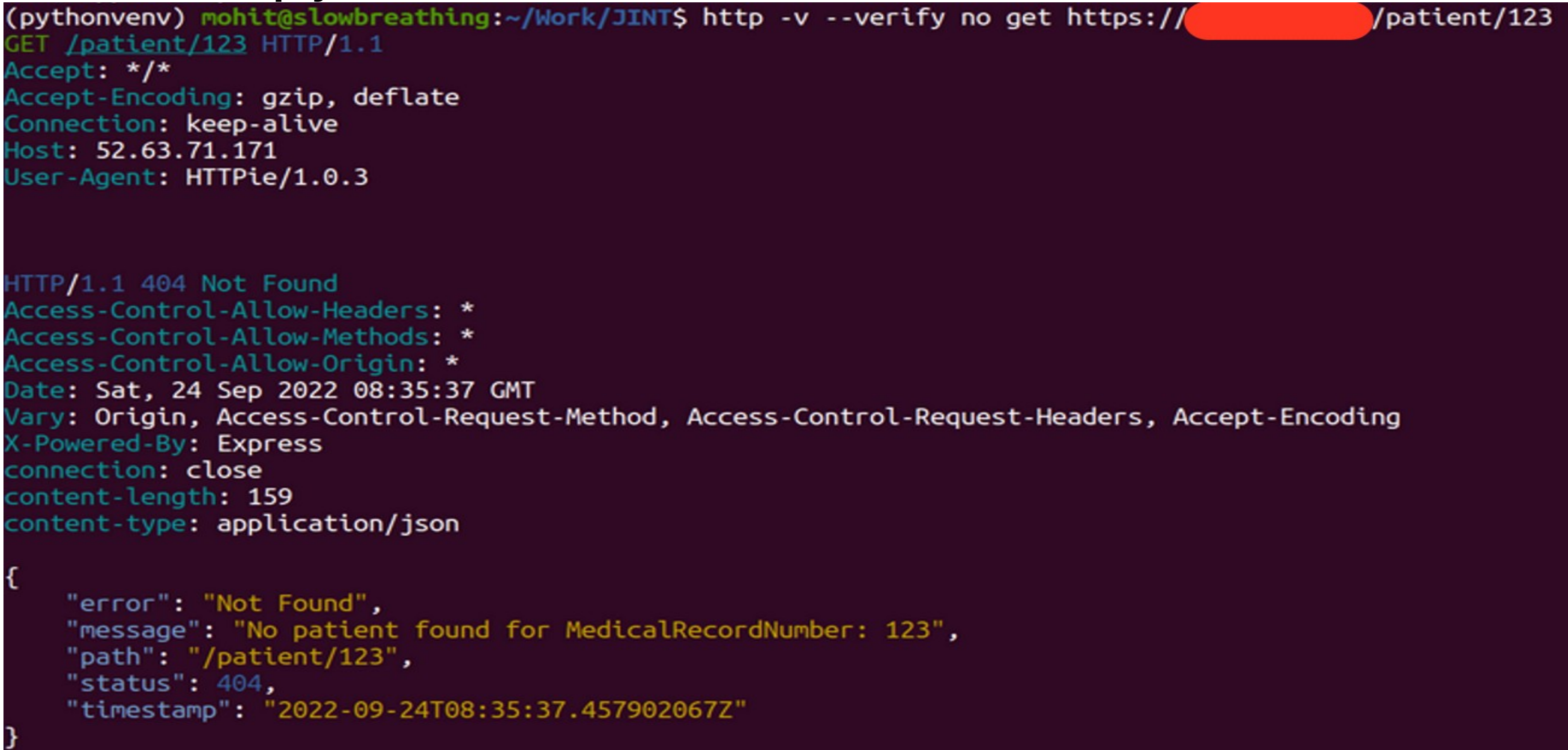
- **SwitchIfEmpty**
  - it is also used to throw exceptions and prevent further processing and converted to standard exceptions especially in case of http



```
@ResponseStatus(NOT_FOUND)
@ExceptionHandler(NotFoundException.class)
public @ResponseBody HttpErrorInfo handleNotFoundExceptions(
  ServerHttpRequest request, NotFoundException ex) {

  return createHttpErrorInfo(NOT_FOUND, request, ex);
}
```

# Operators:Conditional

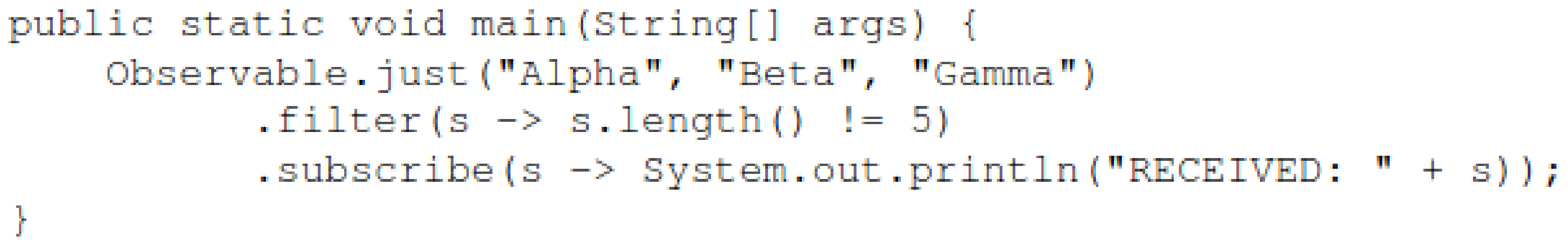
- **SwitchIfEmpty**



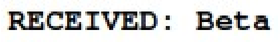
```
(pythonvenv) mohit@slowbreathing:~/Work/JINT$ http -v --verify no get https://[redacted]/patient/123
GET /patient/123 HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: 52.63.71.171
User-Agent: HTTPie/1.0.3



HTTP/1.1 404 Not Found
Access-Control-Allow-Headers: *
Access-Control-Allow-Methods: *
Access-Control-Allow-Origin: *
Date: Sat, 24 Sep 2022 08:35:37 GMT
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers, Accept-Encoding
X-Powered-By: Express
connection: close
content-length: 159
content-type: application/json

{
    "error": "Not Found",
    "message": "No patient found for MedicalRecordNumber: 123",
    "path": "/patient/123",
    "status": 404,
    "timestamp": "2022-09-24T08:35:37.457902067Z"
}
```

## Operators:supressing

- **filter**



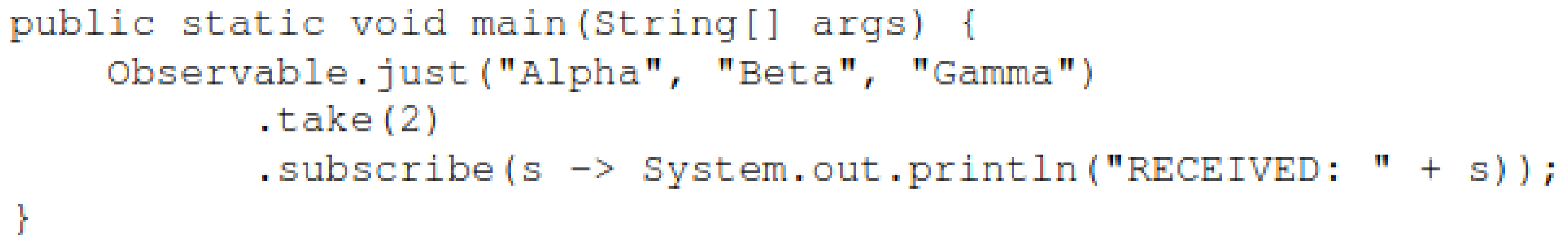
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .filter(s -> s.length() != 5)
            .subscribe(s -> System.out.println("RECEIVED: " + s));
}
```



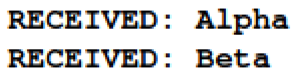
```
RECEIVED: Beta
```

## Operators:supressing

- **take**



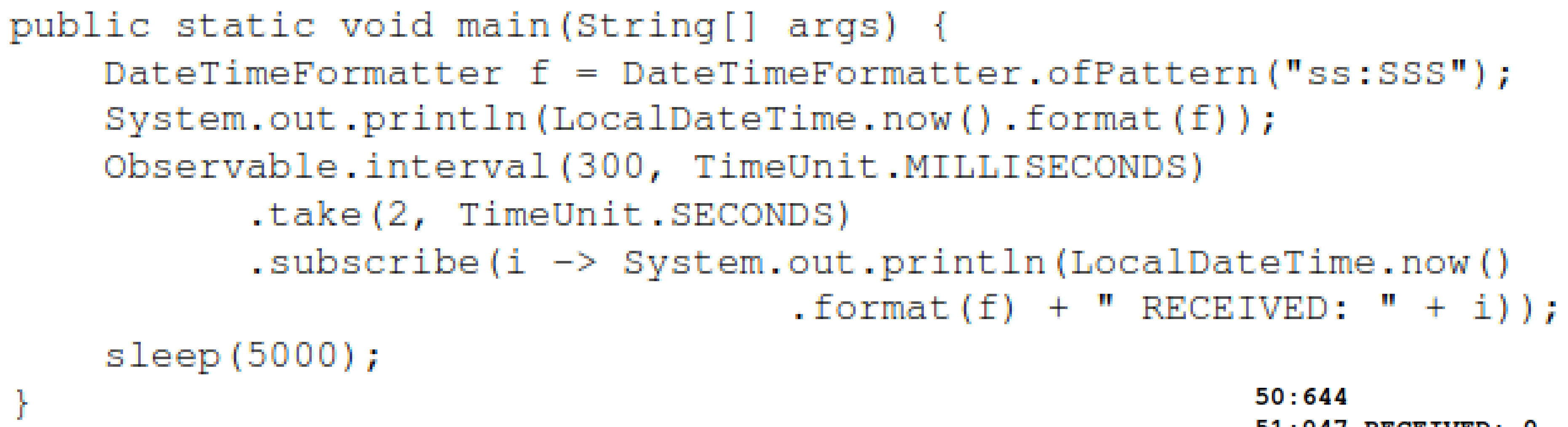
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .take(2)
            .subscribe(s -> System.out.println("RECEIVED: " + s));
}
```



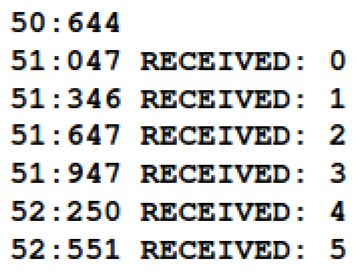
```
RECEIVED: Alpha
RECEIVED: Beta
```

## Operators:supressing

- **take**
  - The other version of the take() operator accepts emissions within the specific time duration and then emits onComplete.



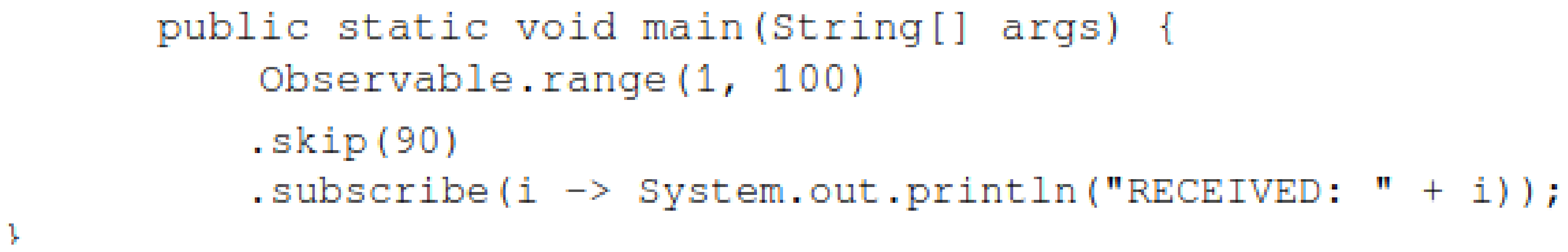
```
public static void main(String[] args) {
    DateTimeFormatter f = DateTimeFormatter.ofPattern("ss:SSS");
    System.out.println(LocalDateTime.now().format(f));
    Observable.interval(300, TimeUnit.MILLISECONDS)
          .take(2, TimeUnit.SECONDS)
          .subscribe(i -> System.out.println(LocalDateTime.now()
                                      .format(f) + " RECEIVED: " + i));
    sleep(5000);
}
```



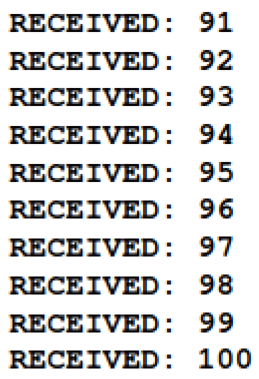
```
50:644
51:047 RECEIVED: 0
51:346 RECEIVED: 1
51:647 RECEIVED: 2
51:947 RECEIVED: 3
52:250 RECEIVED: 4
52:551 RECEIVED: 5
```

# Operators:supressing

- **skip**



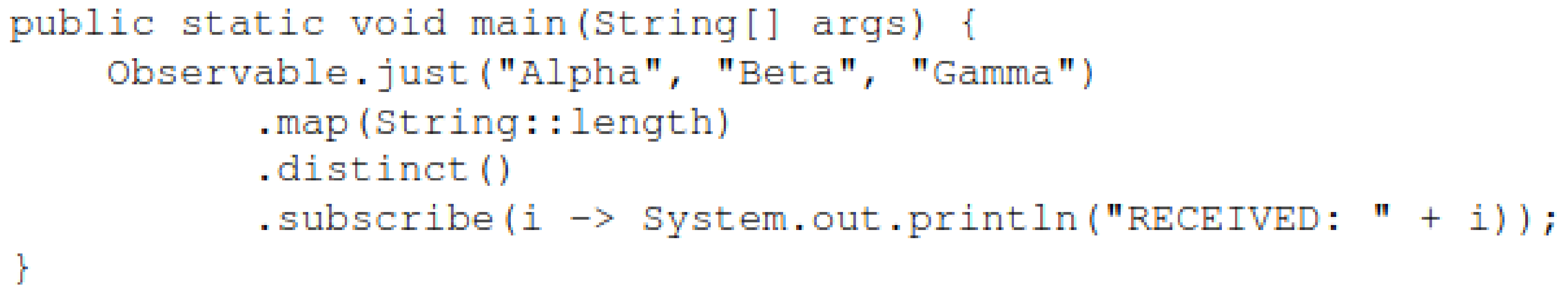
```java
    public static void main(String[] args) {
        Observable.range(1, 100)
        .skip(90)
        .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```



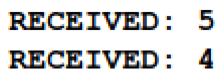
```
RECEIVED: 91
RECEIVED: 92
RECEIVED: 93
RECEIVED: 94
RECEIVED: 95
RECEIVED: 96
RECEIVED: 97
RECEIVED: 98
RECEIVED: 99
RECEIVED: 100
```

# Operators:supressing

- **distinct**



```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .map(String::length)
            .distinct()
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```



```
RECEIVED: 5
RECEIVED: 4
```

## Operators:supressing

- **distinct**

```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .map(String::length)
            .distinct()
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

```
RECEIVED: 5
RECEIVED: 4
```

```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .distinct(String::length)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

```
RECEIVED: Alpha
RECEIVED: Beta
```

## Operators:supressing

- **DistinctUntilChanged**

```java
public static void main(String[] args) {
    Observable.just(1, 1, 1, 2, 2, 3, 3, 2, 1, 1)
            .distinctUntilChanged()
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

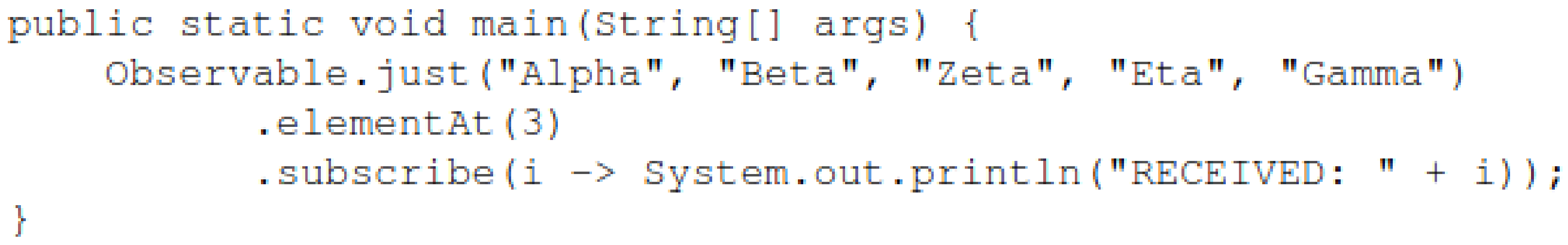
```
RECEIVED: 1
RECEIVED: 2
RECEIVED: 3
RECEIVED: 2
RECEIVED: 1
```

```java
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Zeta", "Eta", "Gamma", "Delta")
            .distinctUntilChanged(String::length)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

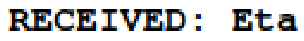
```
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Eta
RECEIVED: Gamma
```

## Operators:supressing

- **elementAt**



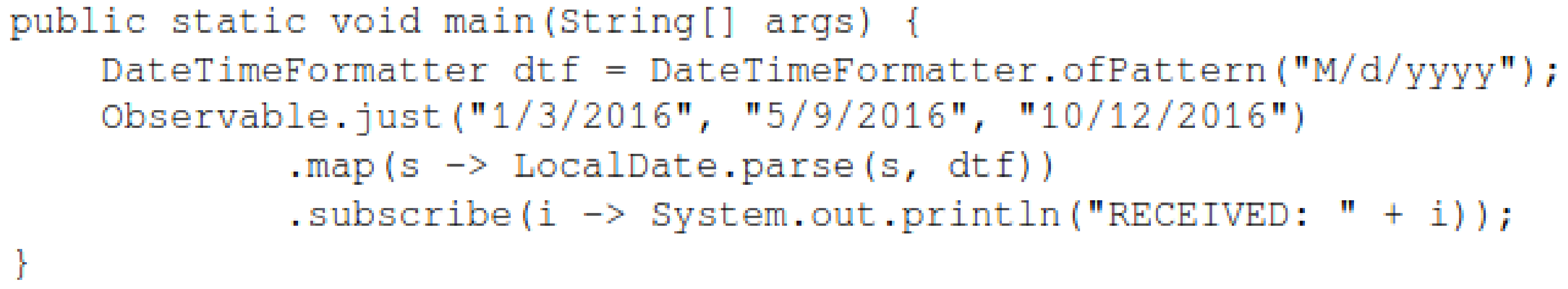
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Zeta", "Eta", "Gamma")
            .elementAt(3)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```



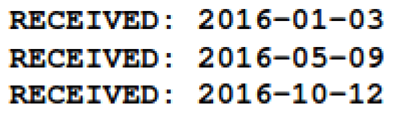
```
RECEIVED: Eta
```

## Operators:transforming

- **map**



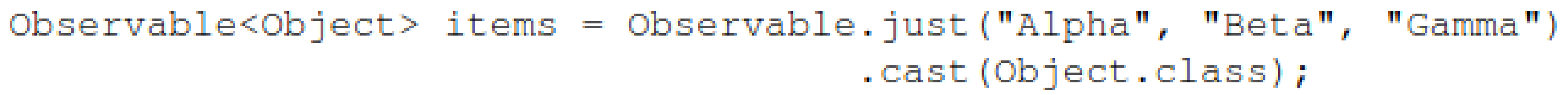
```
public static void main(String[] args) {
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("M/d/yyyy");
    Observable.just("1/3/2016", "5/9/2016", "10/12/2016")
            .map(s -> LocalDate.parse(s, dtf))
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```



```
RECEIVED: 2016-01-03
RECEIVED: 2016-05-09
RECEIVED: 2016-10-12
```

# Operators:transforming

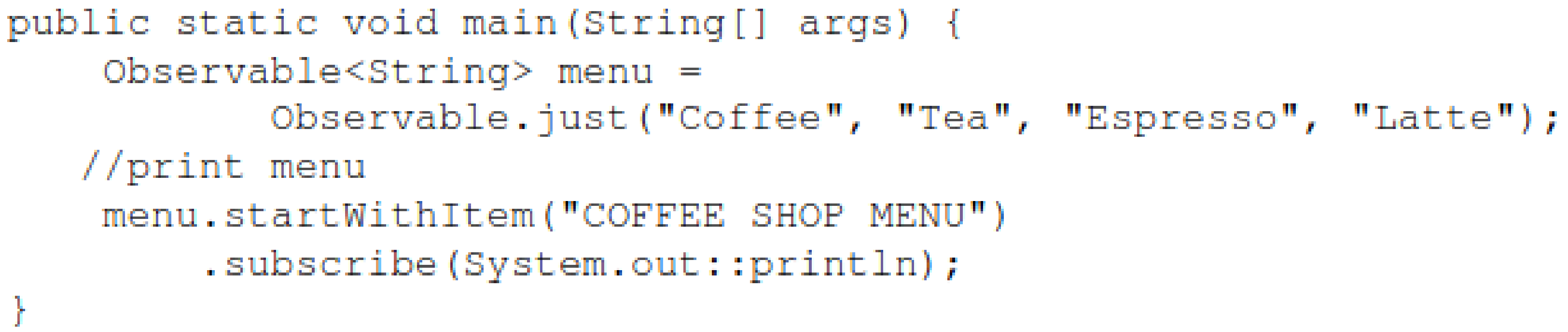
- **Cast**
  - we can use the more specialized shorthand cast(), and simply pass the class type we want to cast to, as shown in this code snippet:



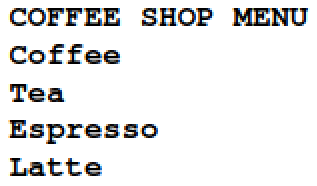
```
Observable<Object> items = Observable.just("Alpha", "Beta", "Gamma")
                                      .cast(Object.class);
```

## Operators:transforming

- **startWithItem**
  - For a given Observable<T>, the startWithItem() operator (previously called startWith() in RxJava 2.x) allows you to insert a value of type T that will be emitted before all the other values.



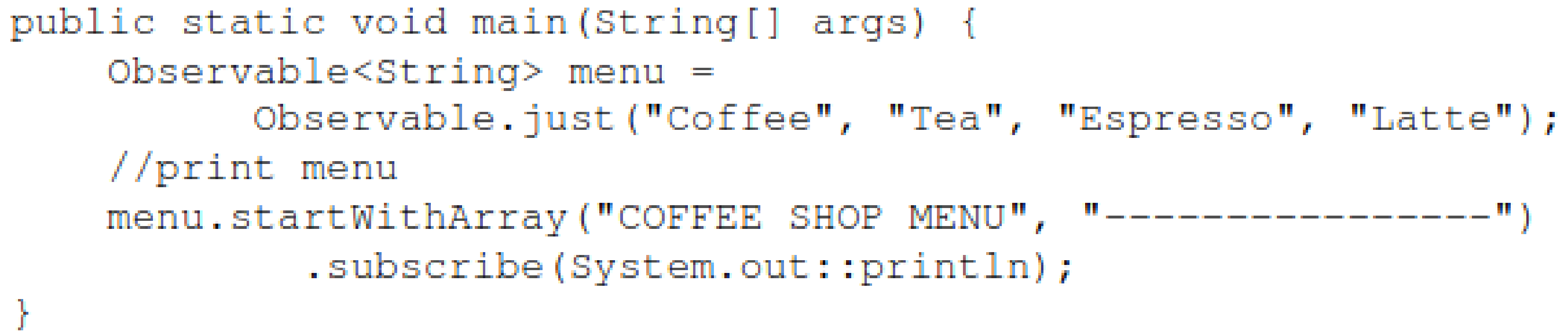
```
public static void main(String[] args) {
    Observable<String> menu =
            Observable.just("Coffee", "Tea", "Espresso", "Latte");
   //print menu
    menu.startWithItem("COFFEE SHOP MENU")
        .subscribe(System.out::println);
}
```



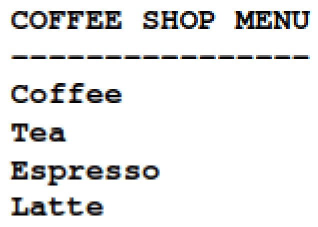
```
COFFEE SHOP MENU
Coffee
Tea
Espresso
Latte
```

## Operators:transforming

- **startWithItem**
  - If you want to start with more than one value emitted first, use startWithArray(), which accepts varargs.



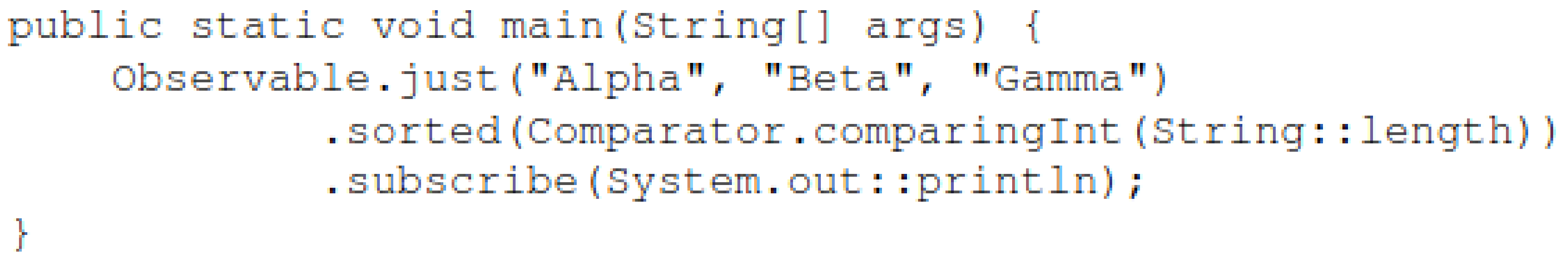
```java
public static void main(String[] args) {
    Observable<String> menu =
          Observable.just("Coffee", "Tea", "Espresso", "Latte");
    //print menu
    menu.startWithArray("COFFEE SHOP MENU", "----------------")
            .subscribe(System.out::println);
}
```



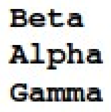
```
COFFEE SHOP MENU
----------------
Coffee
Tea
Espresso
Latte
```

# Operators:transforming

- **Sorted**
  - If you have a finite Observable<T> that emits items that are of a primitive type, String type, or objects that implement Comparable<T>



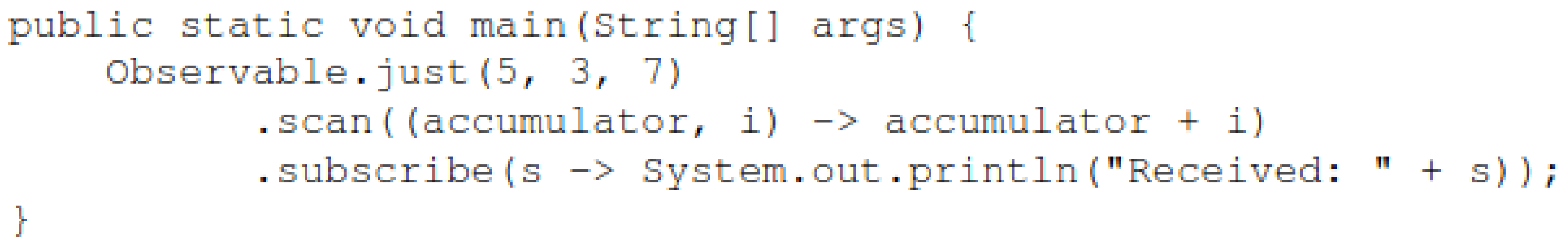
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .sorted(Comparator.comparingInt(String::length))
            .subscribe(System.out::println);
}
```



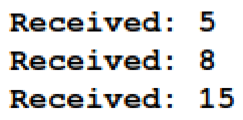
```
Beta
Alpha
Gamma
```

# Operators:transforming

- **scan**
  - The scan() method is a rolling aggregator. It adds every emitted item to the provided accumulator and emits each incremental accumulated value.
    - For instance, let's emit the rolling sum of all of the values emitted so far, including the current one, as follows:



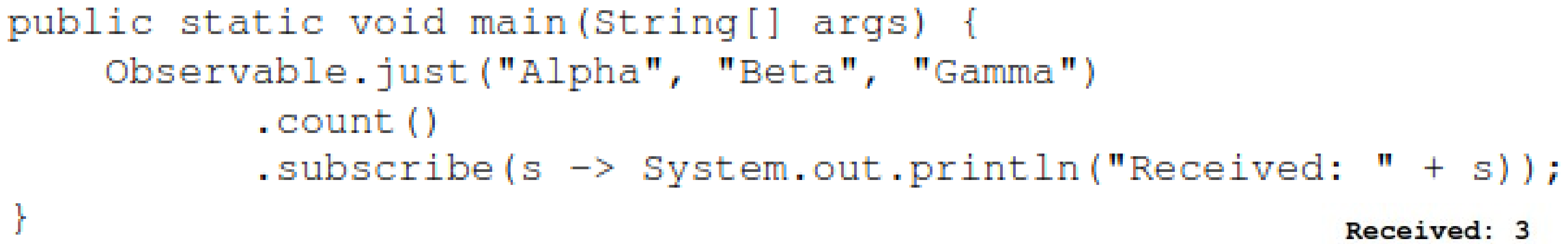
```
public static void main(String[] args) {
    Observable.just(5, 3, 7)
            .scan((accumulator, i) -> accumulator + i)
            .subscribe(s -> System.out.println("Received: " + s));
}
```



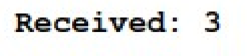
```
Received: 5
Received: 8
Received: 15
```

## Operators:reducing

- **count**
  - The count() operator counts the number of emitted items and emits the result through a Single once onComplete() is called.



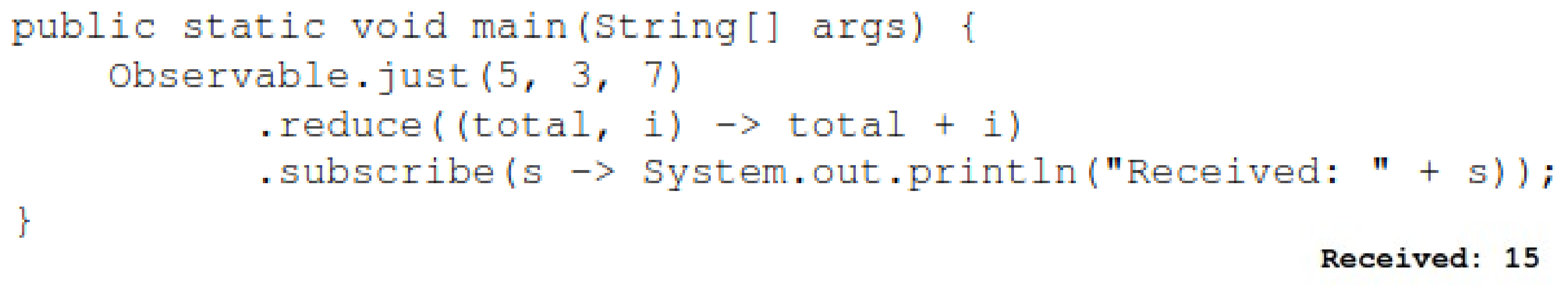
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
          .count()
          .subscribe(s -> System.out.println("Received: " + s));
}
```



```
Received: 3
```

# Operators:reducing

- **reduce**
  - The reduce() operator is syntactically identical to scan(), but it only emits the final result when the source Observable calls onComplete().
    - Depending on which overloaded version is used, it can yield Single or Maybe.

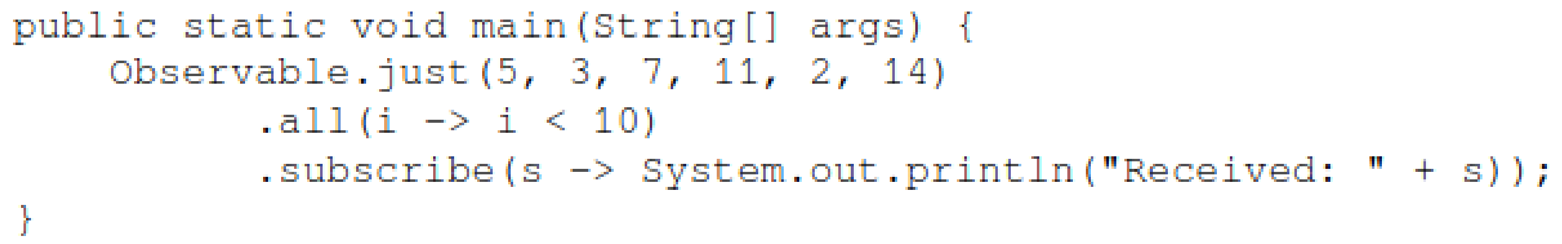


```
public static void main(String[] args) {
    Observable.just(5, 3, 7)
            .reduce((total, i) -> total + i)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

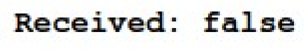
```
Received: 15
```

## Operators:reducing

- **all**
  - The all() operator verifies that all emissions meet the specified criterion and returns a Single<Boolean> object.



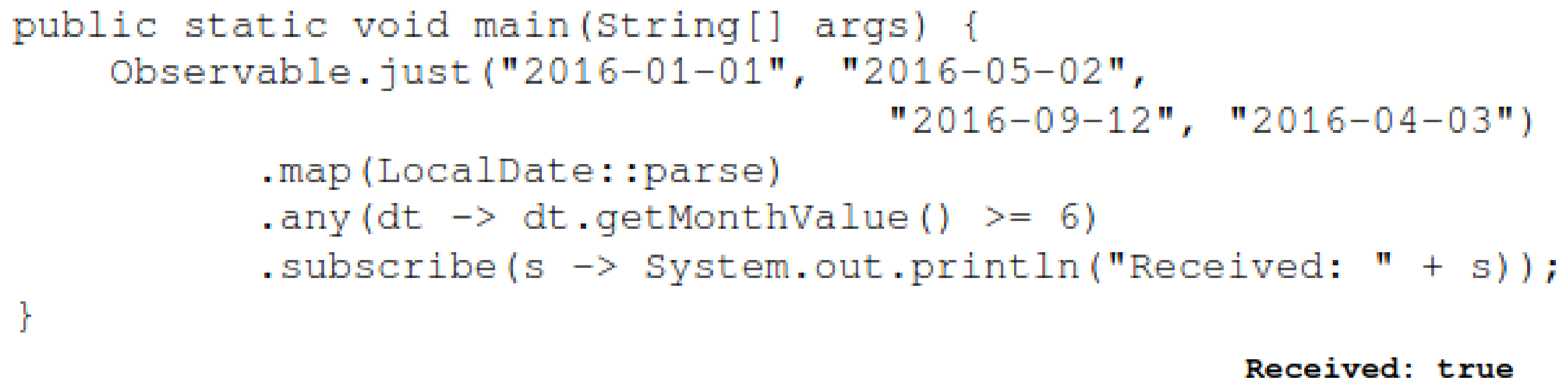
```
public static void main(String[] args) {
    Observable.just(5, 3, 7, 11, 2, 14)
            .all(i -> i < 10)
            .subscribe(s -> System.out.println("Received: " + s));
}
```



```
Received: false
```

# Operators:reducing

- **any**
  - The any() method checks whether at least one emission meets a specified criterion and returns a Single<Boolean>.



```
public static void main(String[] args) {
    Observable.just("2016-01-01", "2016-05-02",
                                           "2016-09-12", "2016-04-03")
            .map(LocalDate::parse)
            .any(dt -> dt.getMonthValue() >= 6)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

```
Received: true
```

## Operators:reducing

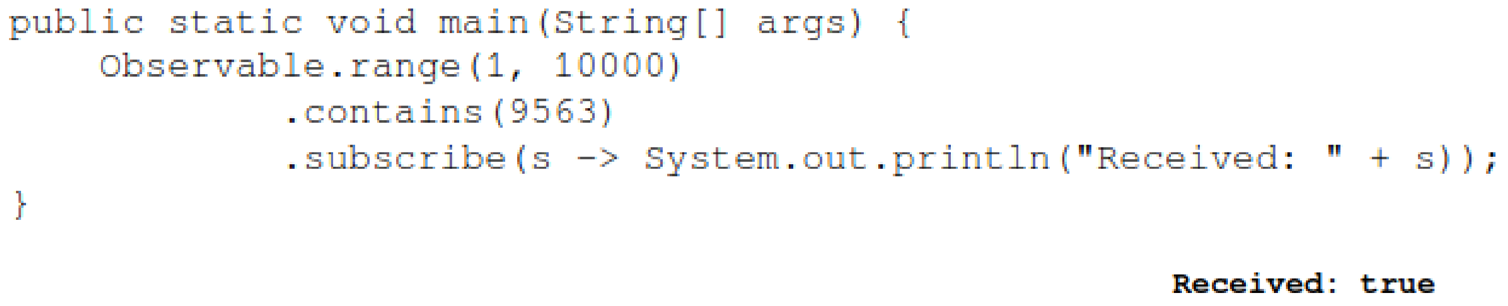
- **isEmpty**
  - The isEmpty() operator checks whether an Observable is going to emit more items.
    - It returns a Single<Boolean> with true if the Observable does not emit items anymore.

```java
public static void main(String[] args) {
    Observable.just("One", "Two", "Three")
            .filter(s -> s.contains("z"))
            .isEmpty()
            .subscribe(s -> System.out.println("Received1: " + s));

    Observable.just("One", "Twoz", "Three")
            .filter(s -> s.contains("z"))
            .isEmpty()
            .subscribe(s -> System.out.println("Received2: " + s));
}
```

```
Received1: true
Received2: false
```

# Operators:reducing

- **contains**
  - The contains() operator checks whether a specified item (based on the hashCode()/equals() implementation) has been emitted by the source Observable.
    - It returns a Single<Boolean> with true if the specified item was emitted, and false if it was not.

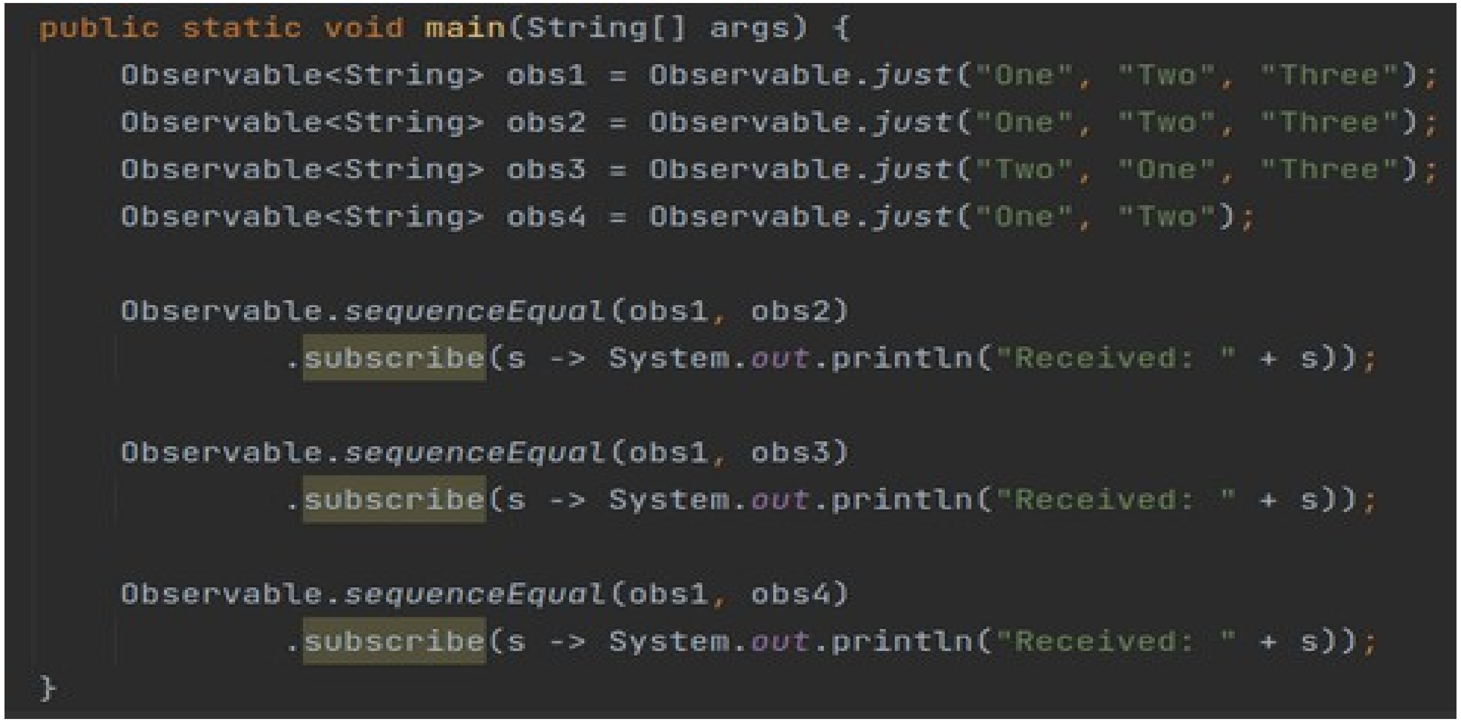


```
public static void main(String[] args) {
    Observable.range(1, 10000)
            .contains(9563)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

```
Received: true
```

# Operators:reducing

- **sequenceEqual**
  - The sequenceEqual() operator checks whether two observables emit the same values in the same order. It returns a Single<Boolean> with true if the emitted sequences are the same pairwise.

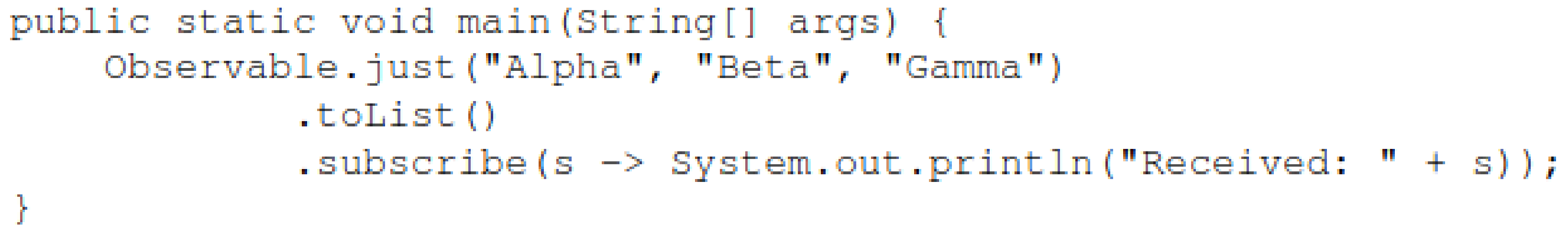


```
public static void main(String[] args) {
    Observable<String> obs1 = Observable.just("One", "Two", "Three");
    Observable<String> obs2 = Observable.just("One", "Two", "Three");
    Observable<String> obs3 = Observable.just("Two", "One", "Three");
    Observable<String> obs4 = Observable.just("One", "Two");

    Observable.sequenceEqual(obs1, obs2)
            .subscribe(s -> System.out.println("Received: " + s));

    Observable.sequenceEqual(obs1, obs3)
            .subscribe(s -> System.out.println("Received: " + s));

    Observable.sequenceEqual(obs1, obs4)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

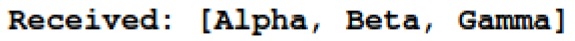
```
Received: true
Received: false
Received: false
```

# Operators:collection

- **toList**
  - The toList() is probably the most often used among all the collection operators. For a given Observable<T>, it collects incoming items into a List<T> and then pushes that List<T> object as a single value through Single<List<T>



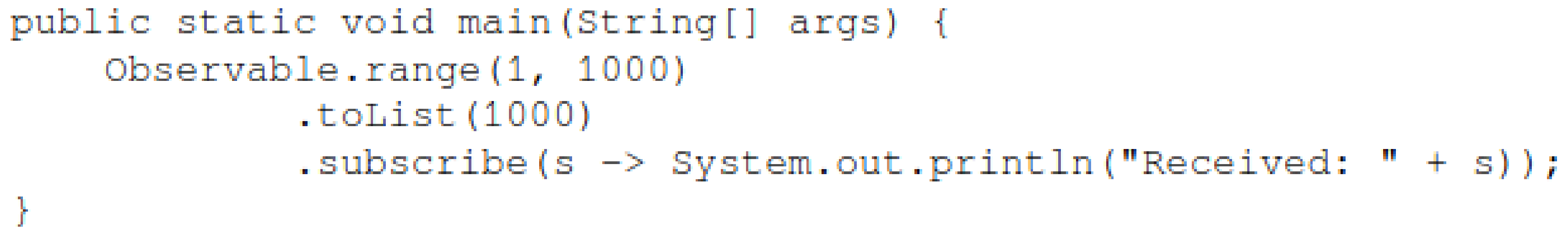
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .toList()
            .subscribe(s -> System.out.println("Received: " + s));
}
```



```
Received: [Alpha, Beta, Gamma]
```

# Operators:collection

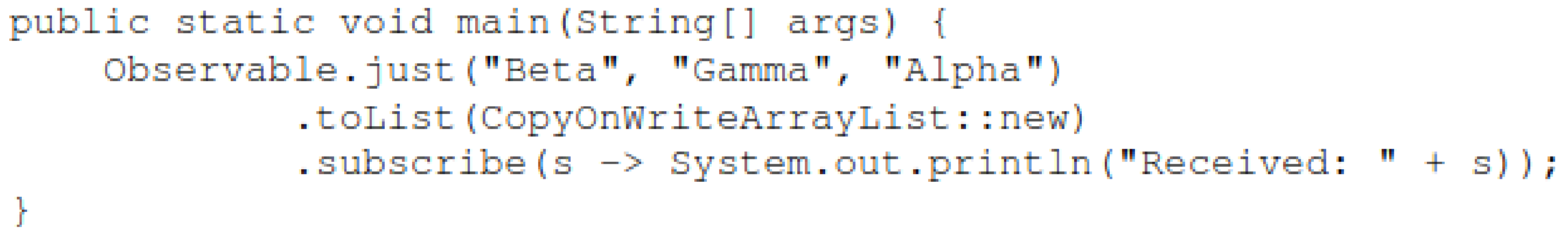
- **toList**
  - By default, toList() uses an ArrayList implementation of the List interface. You can optionally specify an integer argument to serve as the capacityHint value that optimizes the initialization of the ArrayList to expect roughly that number of items



```
public static void main(String[] args) {
    Observable.range(1, 1000)
            .toList(1000)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

# Operators:collection

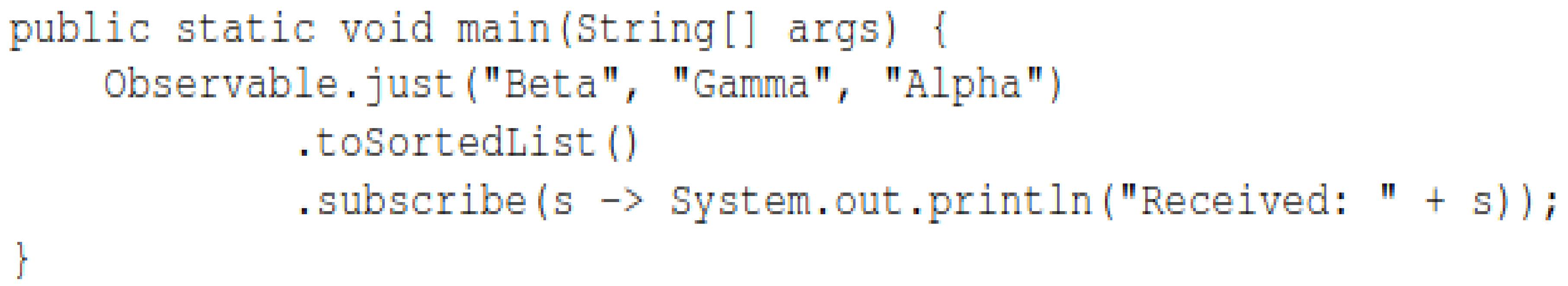
- **toList**
  - If you want to use a different List implementation, you can provide a Callable function as an argument to specify one. In the following code snippet, we provide a CopyOnWriteArrayList instance to serve as a List implementation:



```
public static void main(String[] args) {
    Observable.just("Beta", "Gamma", "Alpha")
            .toList(CopyOnWriteArrayList::new)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

# Operators:collection

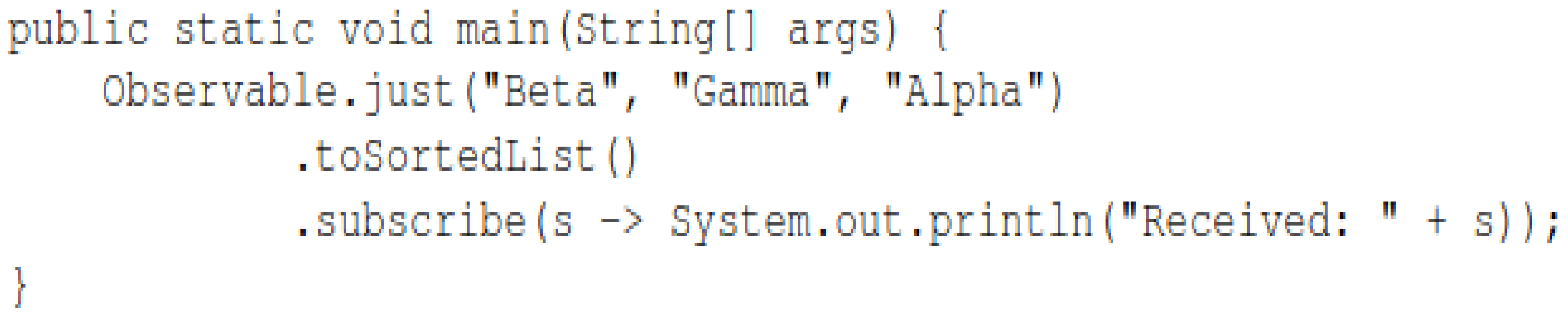
- **toSortedList**
  - A different flavor of toList() operator is toSortedList(). It collects the emitted values into a List object that has the elements sorted in a natural order (based on their Comparable implementation).



```
public static void main(String[] args) {
    Observable.just("Beta", "Gamma", "Alpha")
            .toSortedList()
            .subscribe(s -> System.out.println("Received: " + s));
}
```

# Operators:collection

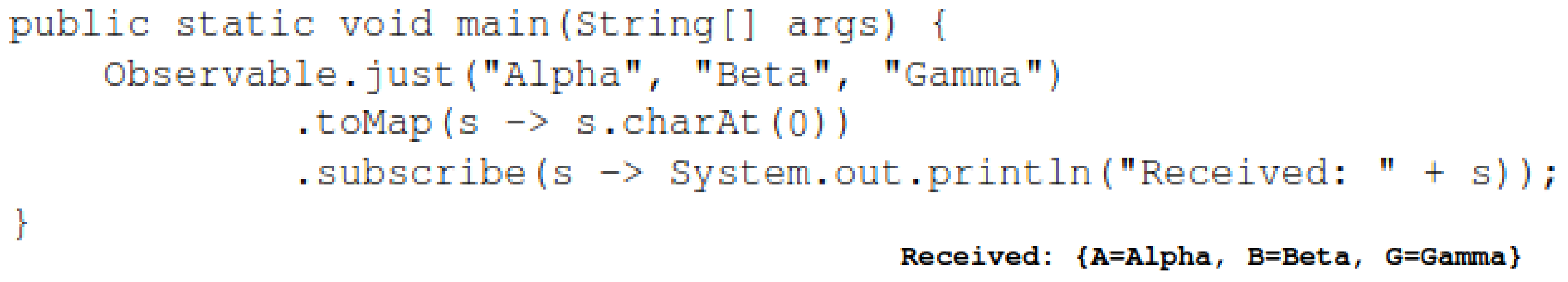
- **toSortedList**
  - A different flavor of toList() operator is toSortedList(). It collects the emitted values into a List object that has the elements sorted in a natural order (based on their Comparable implementation).



```
public static void main(String[] args) {
    Observable.just("Beta", "Gamma", "Alpha")
            .toSortedList()
            .subscribe(s -> System.out.println("Received: " + s));
}
```

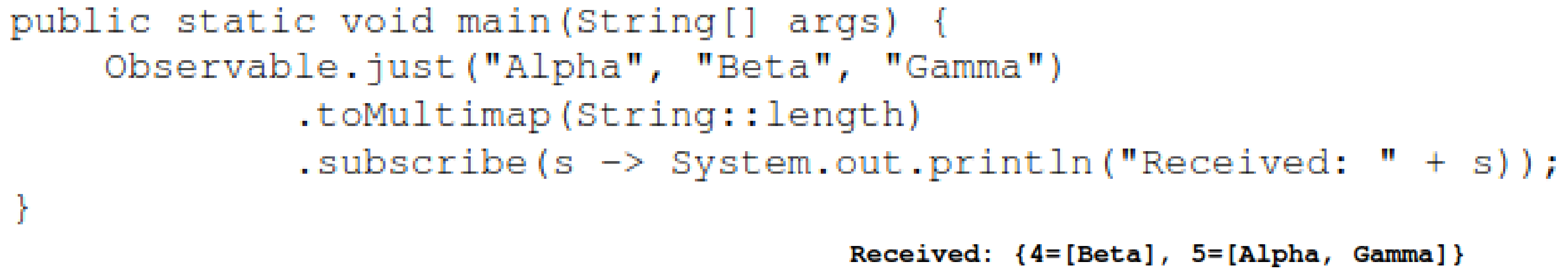
# Operators:collection

- **toMap and toMultiMap**



```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .toMap(s -> s.charAt(0))
            .subscribe(s -> System.out.println("Received: " + s));
}
```

```
Received: {A=Alpha, B=Beta, G=Gamma}
```

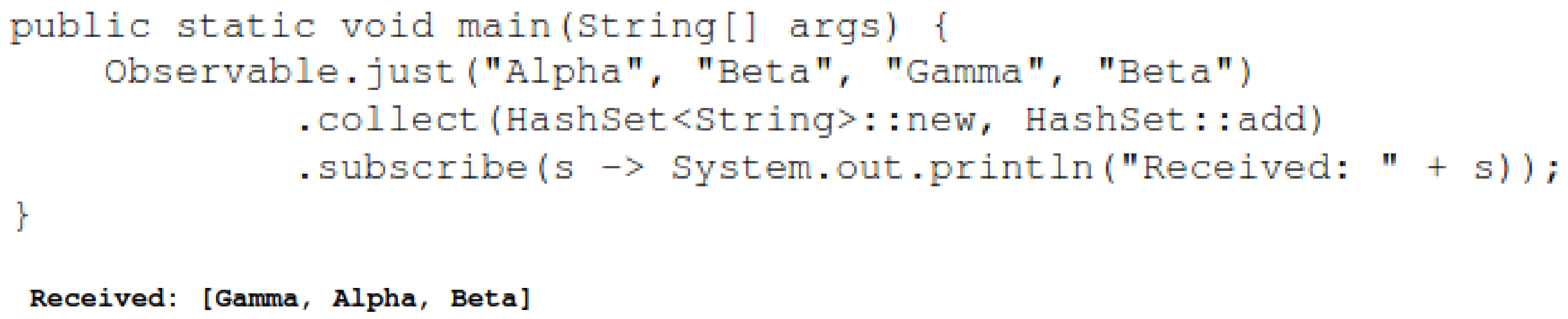


```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .toMultimap(String::length)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

```
Received: {4=[Beta], 5=[Alpha, Gamma]}
```

# Operators:collection

- **Collect**
  - When none of the collection operators can do what you need, you can always use the collect() operator to specify a different type to collect items into.
    - For instance, there is no toSet() operator to collect emissions in a Set<T>

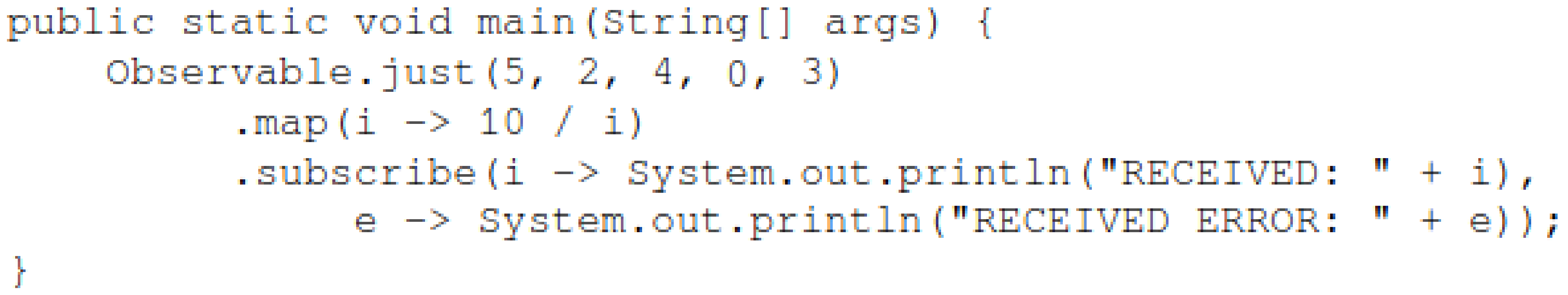


```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma", "Beta")
            .collect(HashSet<String>::new, HashSet::add)
            .subscribe(s -> System.out.println("Received: " + s));
}
```

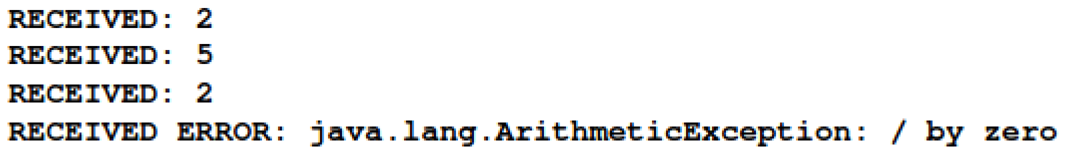
```
Received: [Gamma, Alpha, Beta]
```

# Operators:error

- **Error**
  - Exceptions can occur almost anywhere in the chain of the Observable operators, and we already know about the onError event that is communicated down the Observable chain to the Observer.



```java
public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3)
            .map(i -> 10 / i)
            .subscribe(i -> System.out.println("RECEIVED: " + i),
                    e -> System.out.println("RECEIVED ERROR: " + e));
}
```



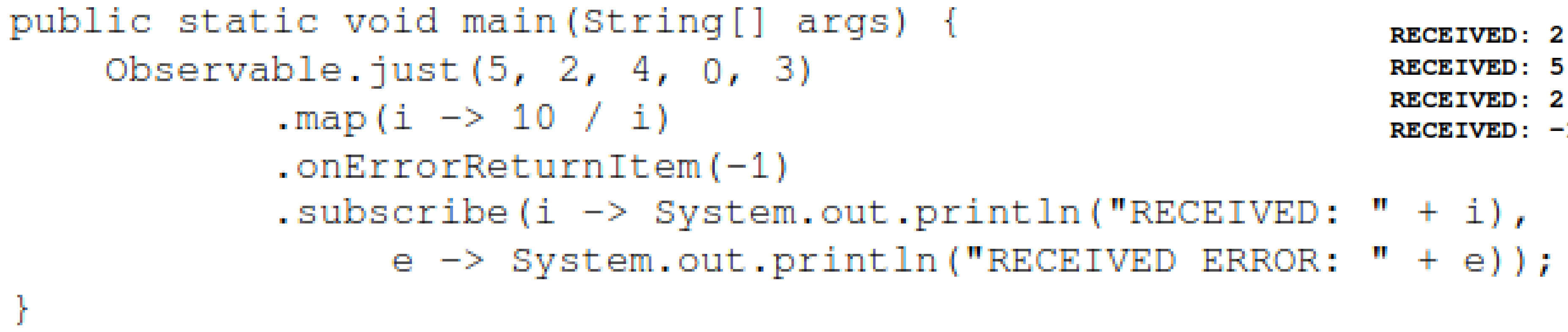
```
RECEIVED: 2
RECEIVED: 5
RECEIVED: 2
RECEIVED ERROR: java.lang.ArithmeticException: / by zero
```

# Operators:error

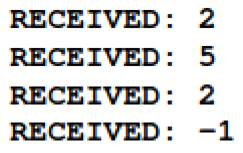
- **Error**
  - But sometimes, we want to **intercept exceptions before they get to the Observer** and attempt some form of recovery.
    - We can also pretend that the error never happened and expect to continue processing the emissions.
    - more productive approach to error handling would be
      - to attempt resubscribing or
      - switch to an alternate source Observable.
    - And if you find that none of the error recovery operators meet your needs, the chances are you can **compose one yourself**.

## Operators:error

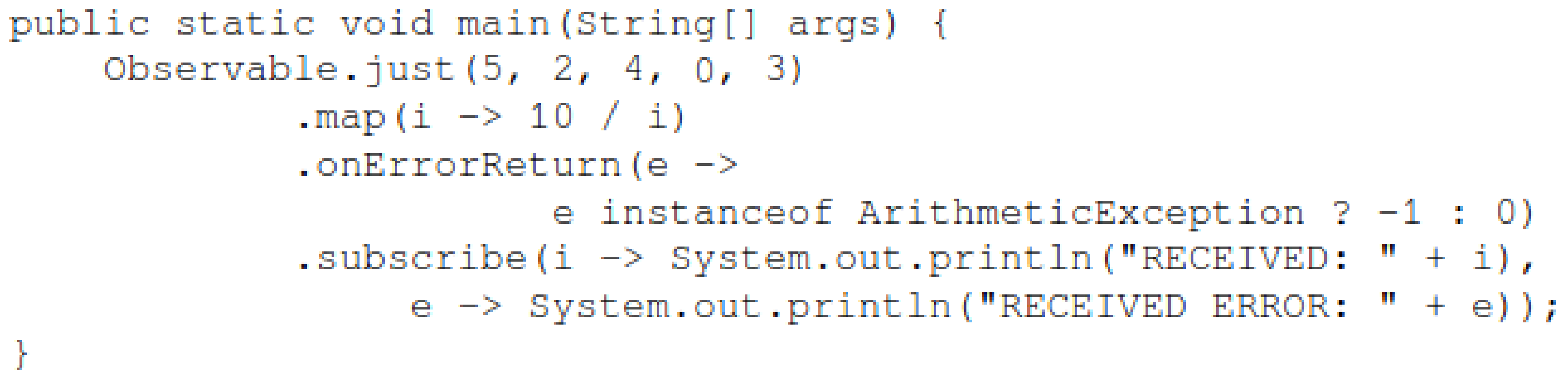
- **onErrorReturnItem and onErrorReturn**



```
public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3)
            .map(i -> 10 / i)
            .onErrorReturnItem(-1)
            .subscribe(i -> System.out.println("RECEIVED: " + i),
                e -> System.out.println("RECEIVED ERROR: " + e));
}
```



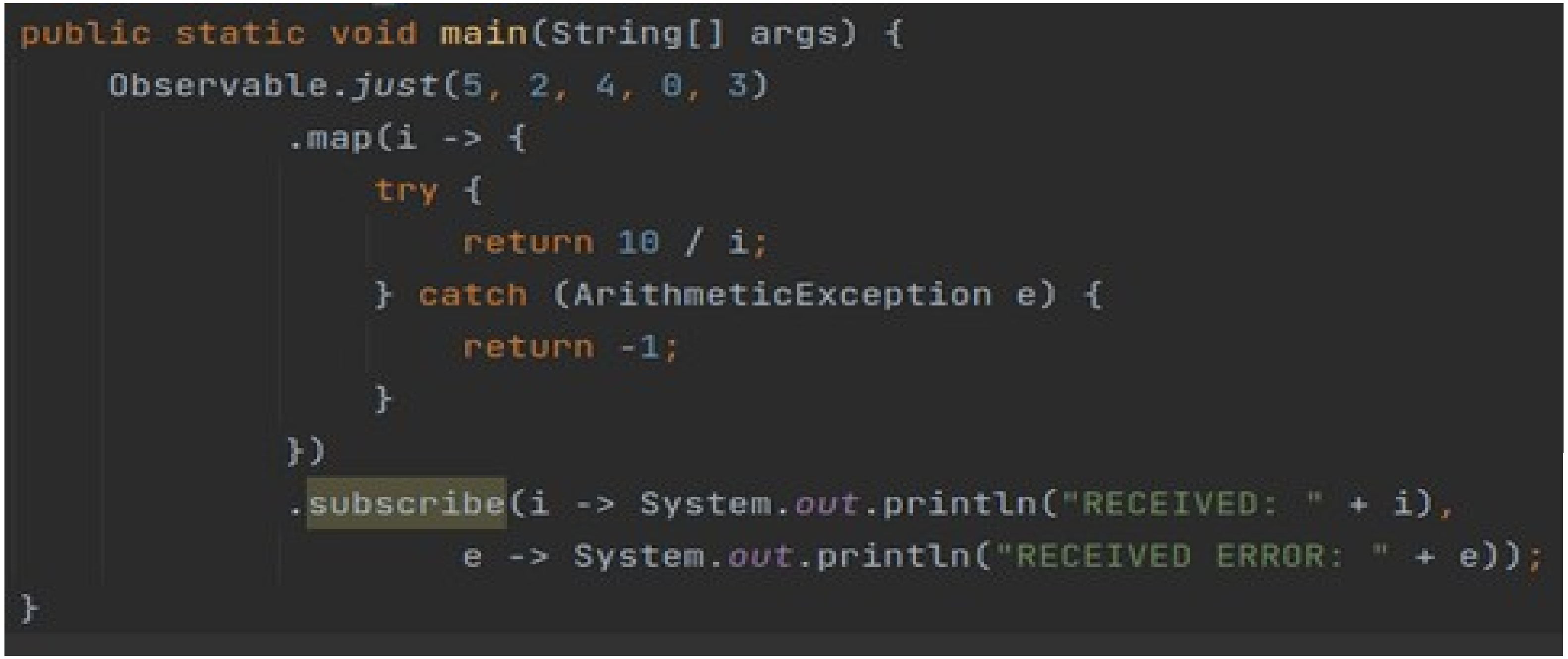
```
RECEIVED: 2
RECEIVED: 5
RECEIVED: 2
RECEIVED: -1
```



```
public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3)
            .map(i -> 10 / i)
            .onErrorReturn(e ->
                    e instanceof ArithmeticException ? -1 : 0)
            .subscribe(i -> System.out.println("RECEIVED: " + i),
                e -> System.out.println("RECEIVED ERROR: " + e));
}
```

# Operators:error

- **map**
  - Although we handled the error, the emission was still terminated after that. We did not get the 3 that was supposed to follow. If you want to resume emissions, you can handle the error within the map() operator where the error occurs.

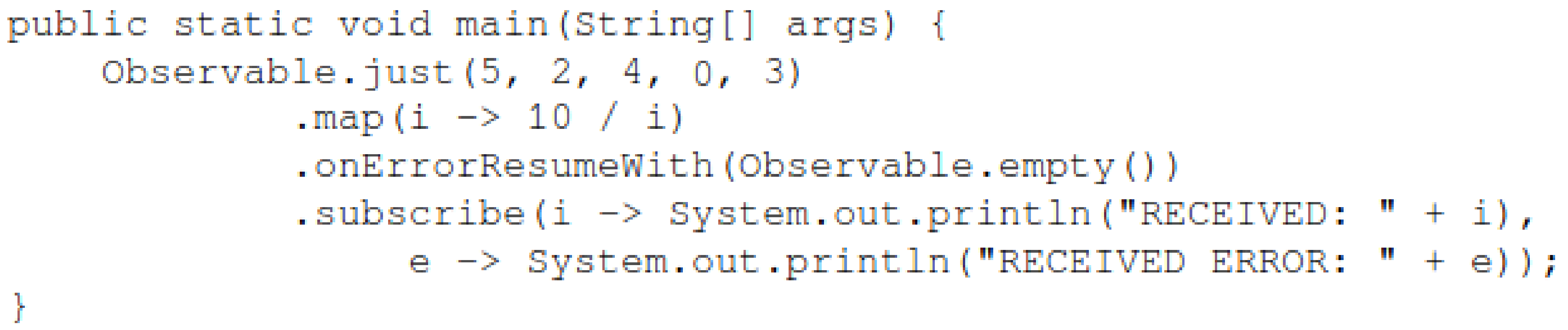


```
public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3)
            .map(i -> {
                try {
                    return 10 / i;
                } catch (ArithmeticException e) {
                    return -1;
                }
            })
            .subscribe(i -> System.out.println("RECEIVED: " + i),
                    e -> System.out.println("RECEIVED ERROR: " + e));
}
```

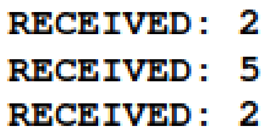
```
RECEIVED: 2
RECEIVED: 5
RECEIVED: 2
RECEIVED: -1
RECEIVED: 3
```

# Operators:error

- **onErrorResumeWith**
  - Although we handled the error, the emission was still terminated after that. We did not get the 3 that was supposed to follow. If you want to resume emissions, you can handle the error within the map() operator where the error occurs.



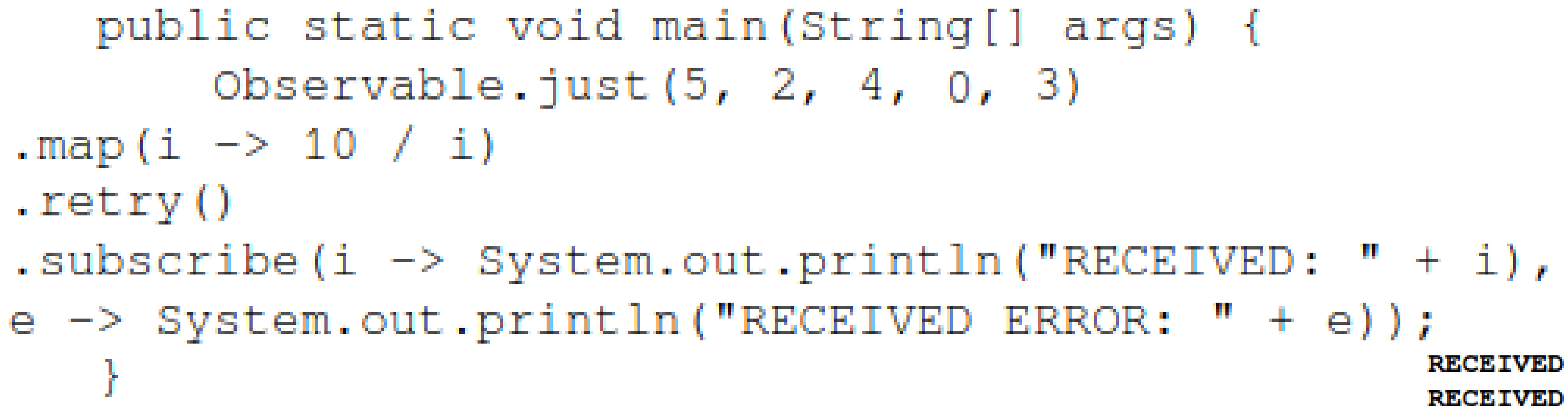
```
public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3)
              .map(i -> 10 / i)
              .onErrorResumeWith(Observable.empty())
              .subscribe(i -> System.out.println("RECEIVED: " + i),
                    e -> System.out.println("RECEIVED ERROR: " + e));
}
```



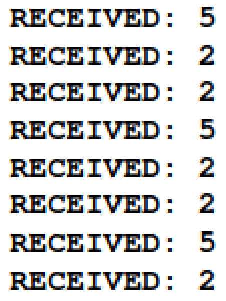
```
RECEIVED: 2
RECEIVED: 5
RECEIVED: 2
```

# Operators:error

- **retry**
  - If you call retry() with no arguments, it will resubscribe an infinite number of times for each error. You need to be careful with retry() without parameters as it can have chaotic effects.



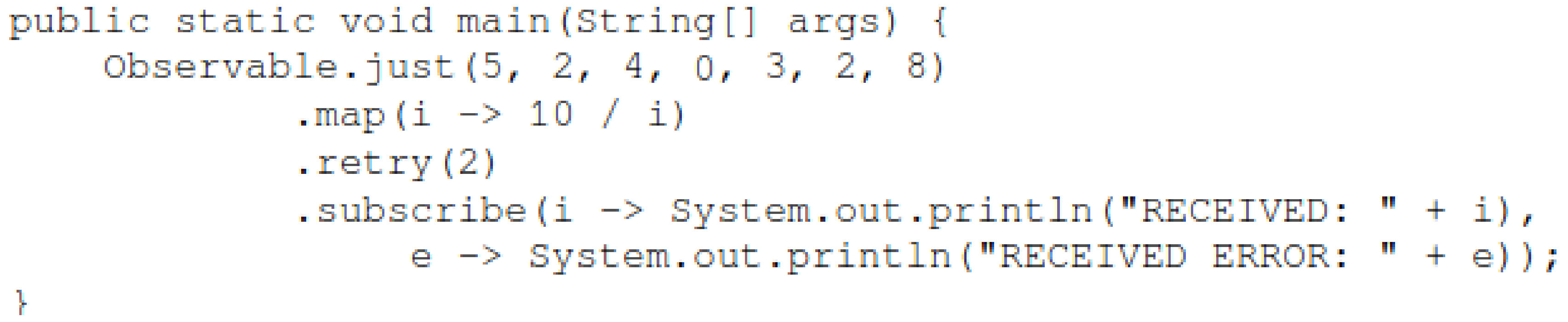
```
    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3)
 .map(i -> 10 / i)
 .retry()
 .subscribe(i -> System.out.println("RECEIVED: " + i),
e -> System.out.println("RECEIVED ERROR: " + e));
    }
```



```
RECEIVED: 5
RECEIVED: 2
RECEIVED: 2
RECEIVED: 5
RECEIVED: 2
RECEIVED: 2
RECEIVED: 5
RECEIVED: 2
```

# Operators:error

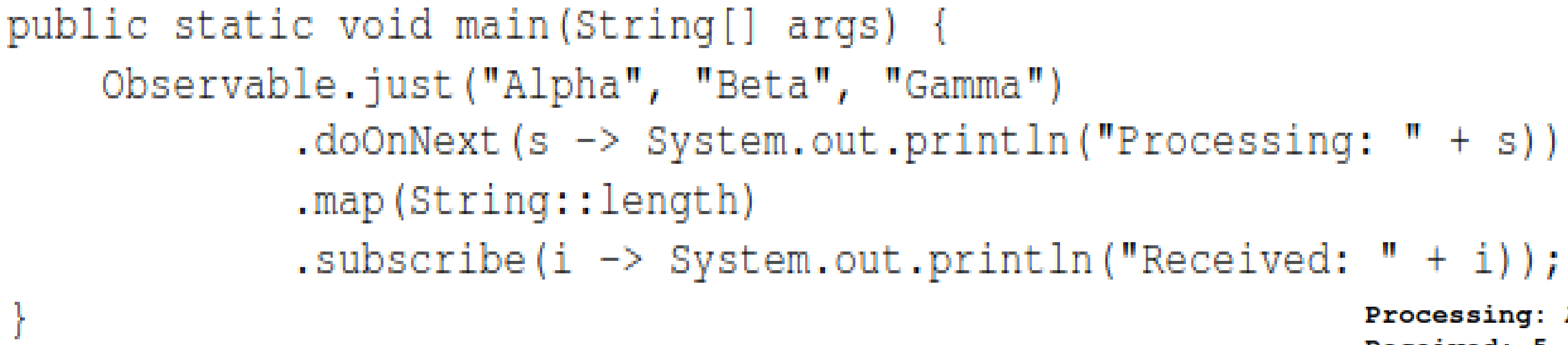
- **retry**
  - It might be safer to specify retry() a fixed number of times before it gives up and just emits the error to the Observer.



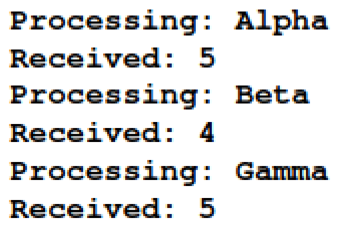
```
public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3, 2, 8)
            .map(i -> 10 / i)
            .retry(2)
            .subscribe(i -> System.out.println("RECEIVED: " + i),
                e -> System.out.println("RECEIVED ERROR: " + e));
}
```

# Operators:action

- **doOnNext() and doAfterNext()**
  - The three operators, doOnNext(), doOnComplete(), and doOnError(), are like putting a mini Observer right in the middle of the Observable chain.
  - The doOnNext() operator allows a peek at each received value before letting it flow into the next operator.



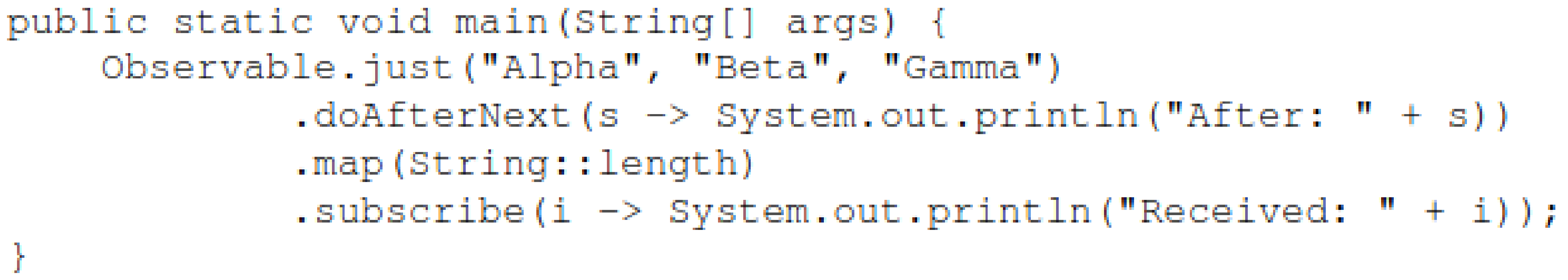
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .doOnNext(s -> System.out.println("Processing: " + s))
            .map(String::length)
            .subscribe(i -> System.out.println("Received: " + i));
}
```



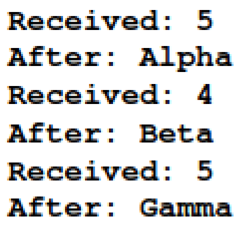
```
Processing: Alpha
Received: 5
Processing: Beta
Received: 4
Processing: Gamma
Received: 5
```

# Operators:action

- **doOnNext() and doAfterNext()**
  - The three operators, doOnNext(), doOnComplete(), and doOnError(), are like putting a mini Observer right in the middle of the Observable chain.
  - The doOnNext() operator allows a peek at each received value before letting it flow into the next operator.



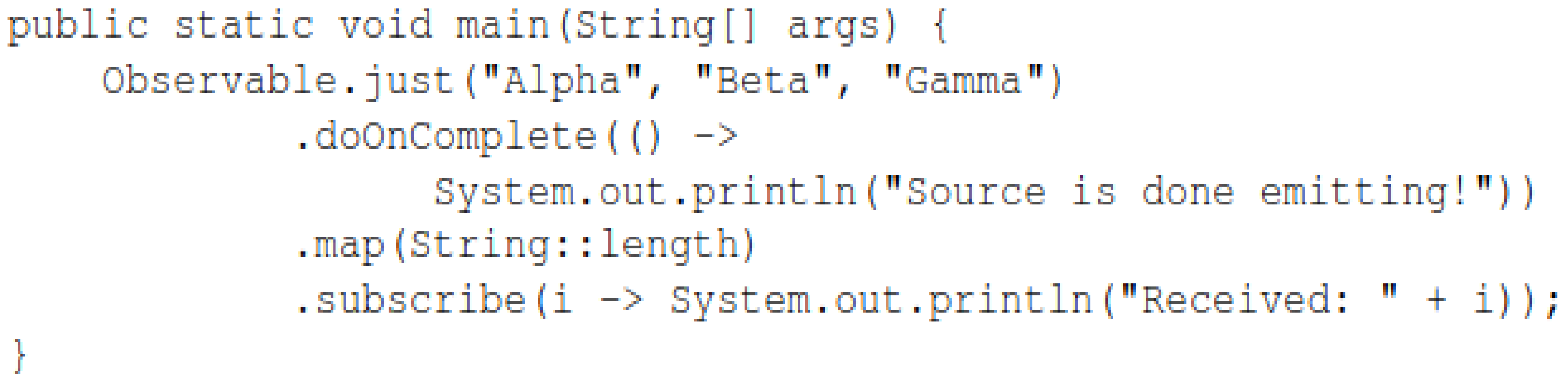
```java
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .doAfterNext(s -> System.out.println("After: " + s))
            .map(String::length)
            .subscribe(i -> System.out.println("Received: " + i));
}
```



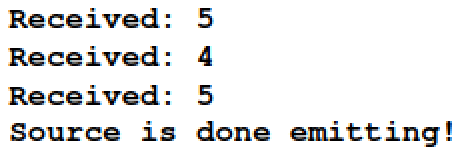
```
Received: 5
After: Alpha
Received: 4
After: Beta
Received: 5
After: Gamma
```

# Operators:action

- **doOnComplete**
  - The onComplete() operator allows you to fire off an action when an onComplete event is emitted at the point in the Observable chain.



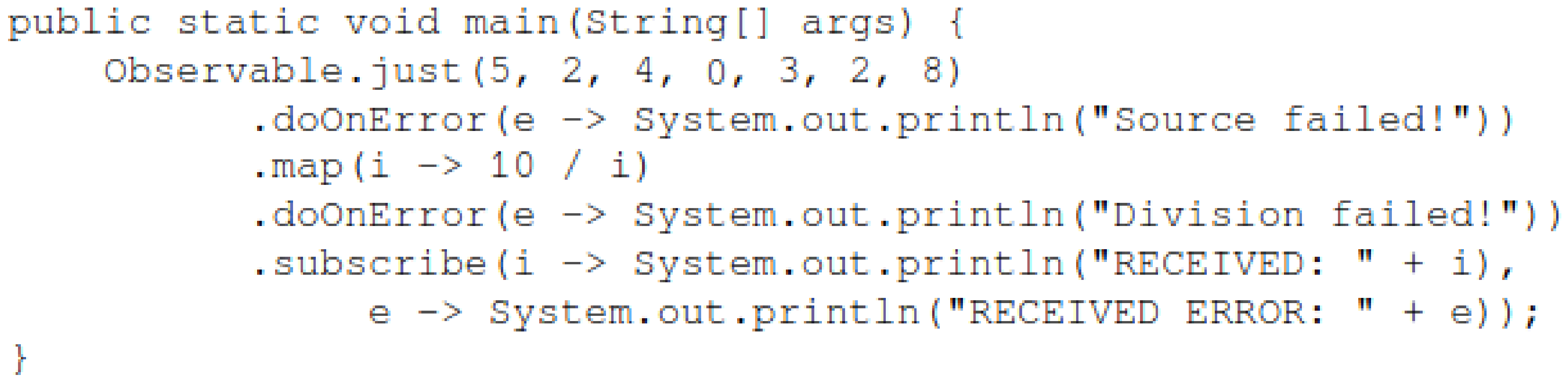
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .doOnComplete(() ->
                    System.out.println("Source is done emitting!"))
            .map(String::length)
            .subscribe(i -> System.out.println("Received: " + i));
}
```



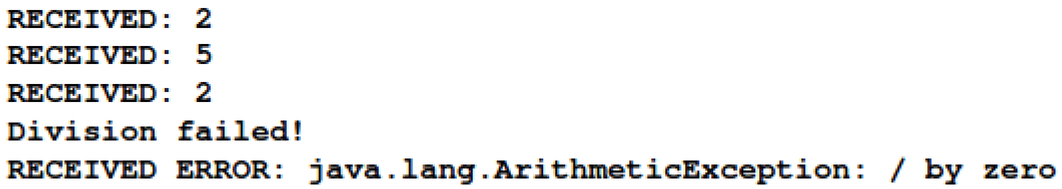
```
Received: 5
Received: 4
Received: 5
Source is done emitting!
```

## Operators:action

- **doOnError**
  - And, of course, onError() will peek at the error being emitted up the chain, and you can perform an action with it. T



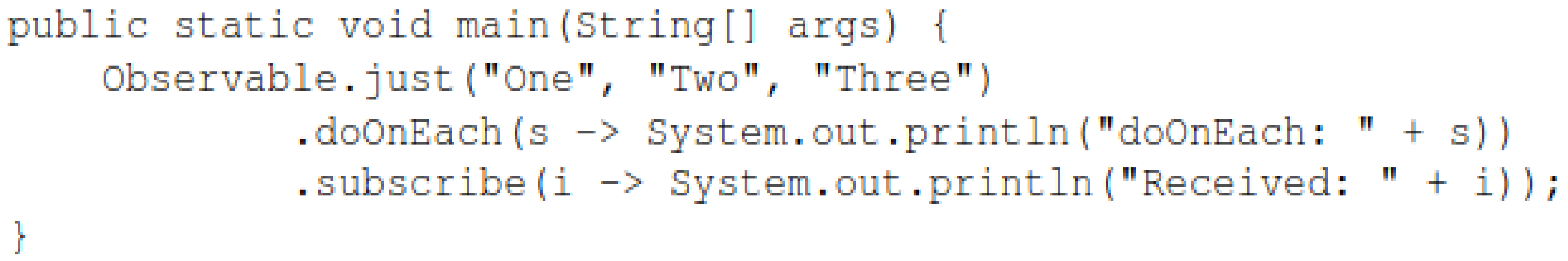
```
public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3, 2, 8)
            .doOnError(e -> System.out.println("Source failed!"))
            .map(i -> 10 / i)
            .doOnError(e -> System.out.println("Division failed!"))
            .subscribe(i -> System.out.println("RECEIVED: " + i),
                e -> System.out.println("RECEIVED ERROR: " + e));
}
```



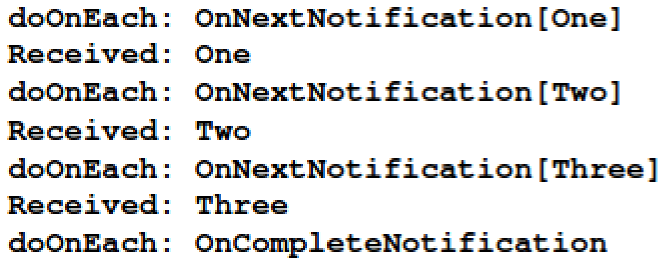
```
RECEIVED: 2
RECEIVED: 5
RECEIVED: 2
Division failed!
RECEIVED ERROR: java.lang.ArithmeticException: / by zero
```

## Operators:action

- **doOnEach**
  - The doOnEach() operator is very similar to doOnNext(). The only difference is that in doOnEach(), the emitted item comes wrapped inside a Notification that also contains the type of the event.



```
public static void main(String[] args) {
    Observable.just("One", "Two", "Three")
            .doOnEach(s -> System.out.println("doOnEach: " + s))
            .subscribe(i -> System.out.println("Received: " + i));
}
```



```
doOnEach: OnNextNotification[One]
Received: One
doOnEach: OnNextNotification[Two]
Received: Two
doOnEach: OnNextNotification[Three]
Received: Three
doOnEach: OnCompleteNotification
```

## Operators:action

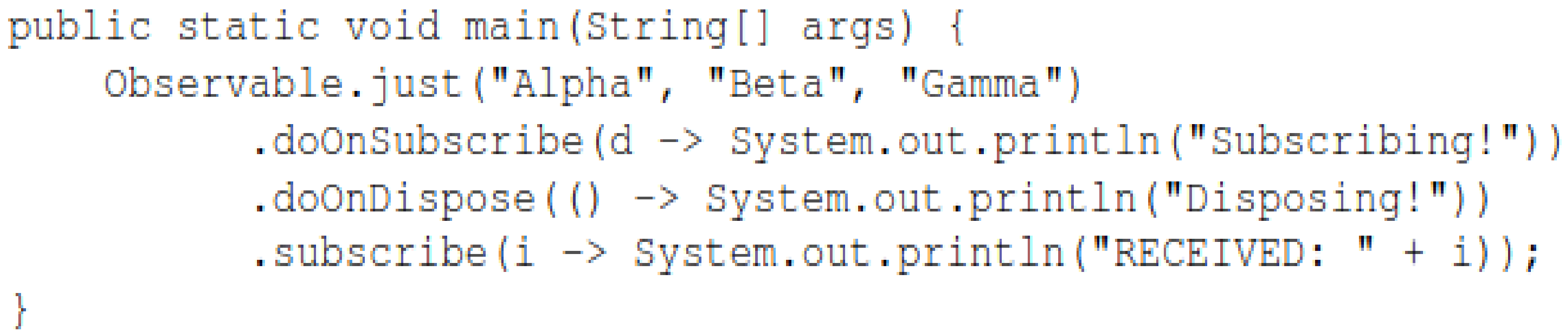
- **doOnEach**
  - The event is wrapped inside OnNextNotification in this case. You can check the event type

```
public static void main(String[] args) {
    Observable.just("One", "Two", "Three")
            .doOnEach(s -> System.out.println("doOnEach: " +
                           s.isOnNext() + ", " + s.isOnError() +
                                    ", " + s.isOnComplete()))
            .subscribe(i -> System.out.println("Received: " + i));
}
```

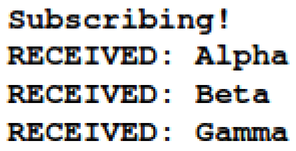
```
doOnEach: true, false, false
Received: One
doOnEach: true, false, false
Received: Two
doOnEach: true, false, false
Received: Three
doOnEach: false, false, true
```

# Operators:action

- **doOnSubscribe and doOnDispose**
  - Set the subscribe event to fire off first, but doOnDispose() was not called. That is because the dispose() method was not called



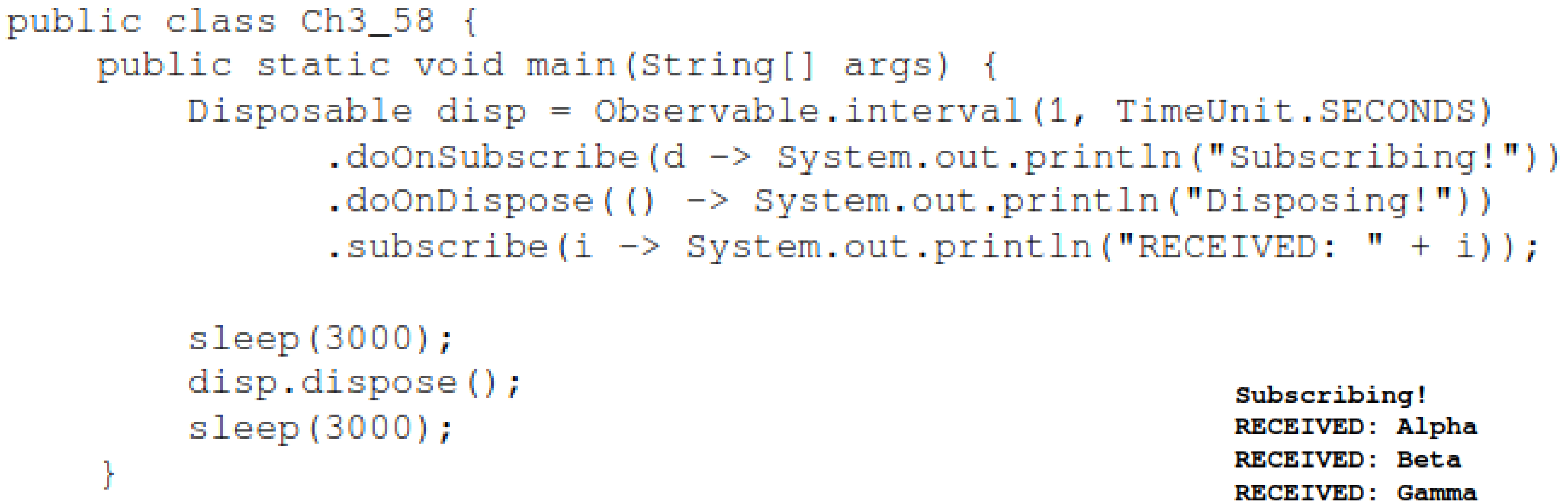
```java
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .doOnSubscribe(d -> System.out.println("Subscribing!"))
            .doOnDispose(() -> System.out.println("Disposing!"))
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```



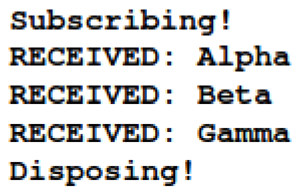
```
Subscribing!
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Gamma
```

# Operators:action

- **doOnSubscribe and doOnDispose**
  - Set the subscribe event to fire off first, but doOnDispose() was not called. That is because the dispose() method was not called



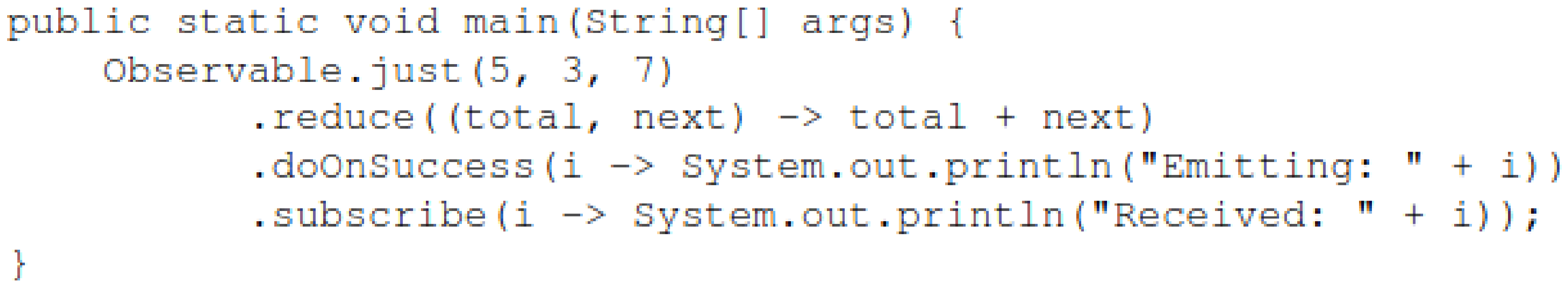
```
public class Ch3_58 {
    public static void main(String[] args) {
        Disposable disp = Observable.interval(1, TimeUnit.SECONDS)
                .doOnSubscribe(d -> System.out.println("Subscribing!"))
                .doOnDispose(() -> System.out.println("Disposing!"))
                .subscribe(i -> System.out.println("RECEIVED: " + i));

        sleep(3000);
        disp.dispose();
        sleep(3000);
    }
```



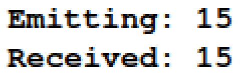
```
Subscribing!
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Gamma
Disposing!
```

## Operators:action

- **doOnSuccess**
  - Remember that Maybe and Single types do not have an onNext() event, but rather an onSuccess() operator to pass a single emission.



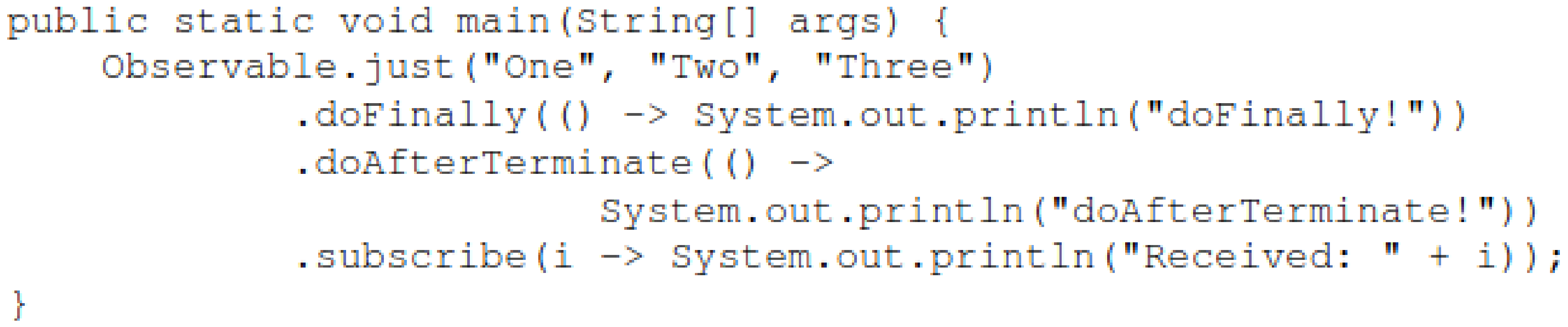
```
public static void main(String[] args) {
    Observable.just(5, 3, 7)
            .reduce((total, next) -> total + next)
            .doOnSuccess(i -> System.out.println("Emitting: " + i))
            .subscribe(i -> System.out.println("Received: " + i));
}
```



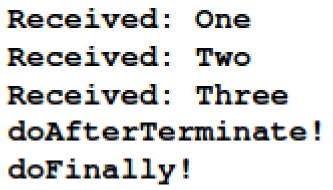
```
Emitting: 15
Received: 15
```

## Operators:action

- **doFinally**
  - The doFinally() operator is executed when onComplete(), onError(), or disposal happens. It is executed under the same conditions as doAfterTerminate(), plus it is also executed after the disposal.



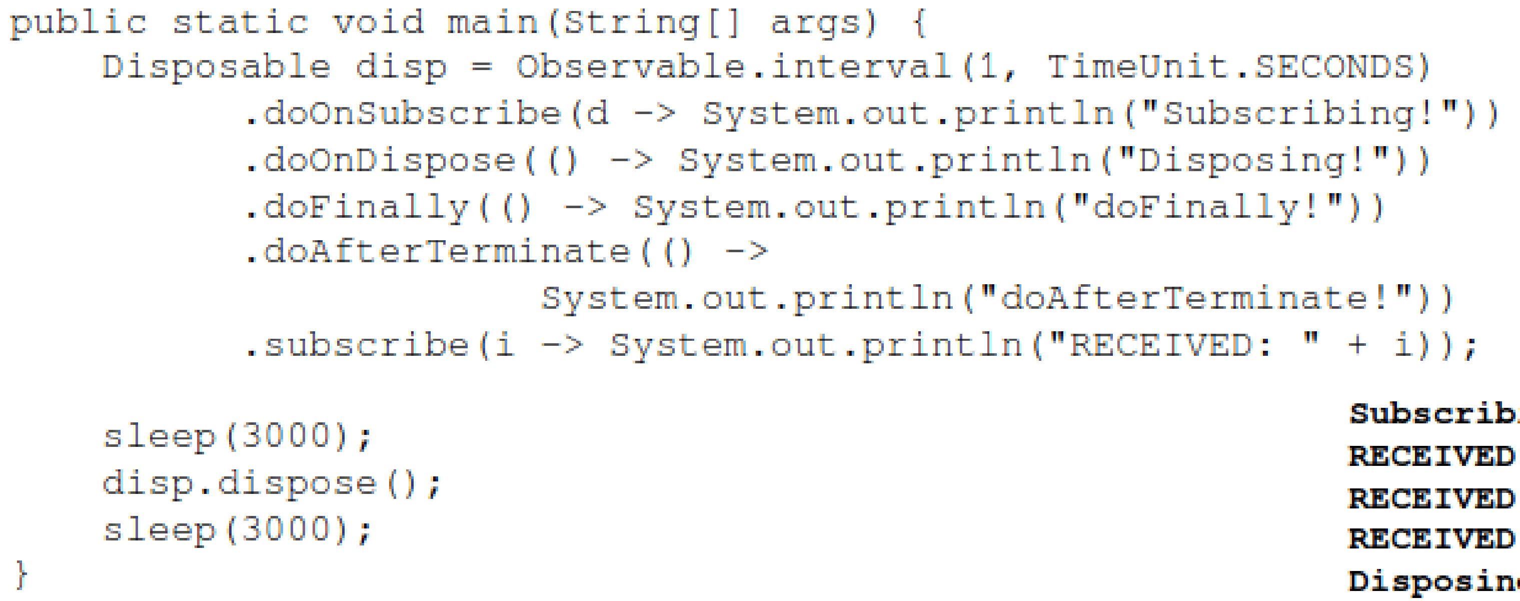
```
public static void main(String[] args) {
    Observable.just("One", "Two", "Three")
            .doFinally(() -> System.out.println("doFinally!"))
            .doAfterTerminate(() ->
                    System.out.println("doAfterTerminate!"))
            .subscribe(i -> System.out.println("Received: " + i));
}
```



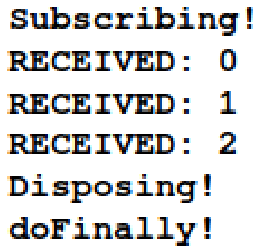
```
Received: One
Received: Two
Received: Three
doAfterTerminate!
doFinally!
```

# Operators:action

- **doFinally**
  - How they work when dispose() is called.
  - The location of these operators in the chain does not matter, because they are driven by the events, not by the emitted data.



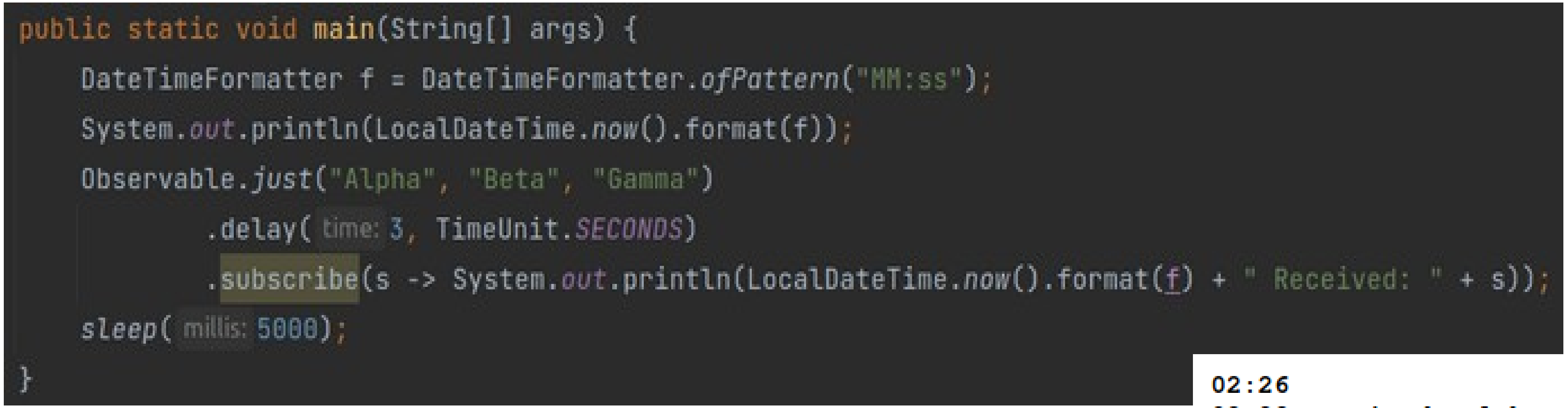
```
public static void main(String[] args) {
    Disposable disp = Observable.interval(1, TimeUnit.SECONDS)
            .doOnSubscribe(d -> System.out.println("Subscribing!"))
            .doOnDispose(() -> System.out.println("Disposing!"))
            .doFinally(() -> System.out.println("doFinally!"))
            .doAfterTerminate(() ->
                        System.out.println("doAfterTerminate!"))
            .subscribe(i -> System.out.println("RECEIVED: " + i));

    sleep(3000);
    disp.dispose();
    sleep(3000);
}
```



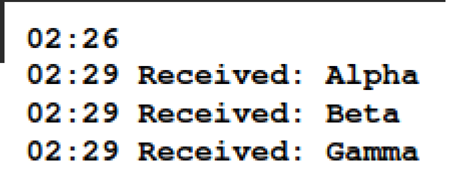
```
Subscribing!
RECEIVED: 0
RECEIVED: 1
RECEIVED: 2
Disposing!
doFinally!
```

# Operators:utility

- **delay**
  - Postpone emissions using the delay() operator. It will hold any received emissions and delay each one for the specified time period.



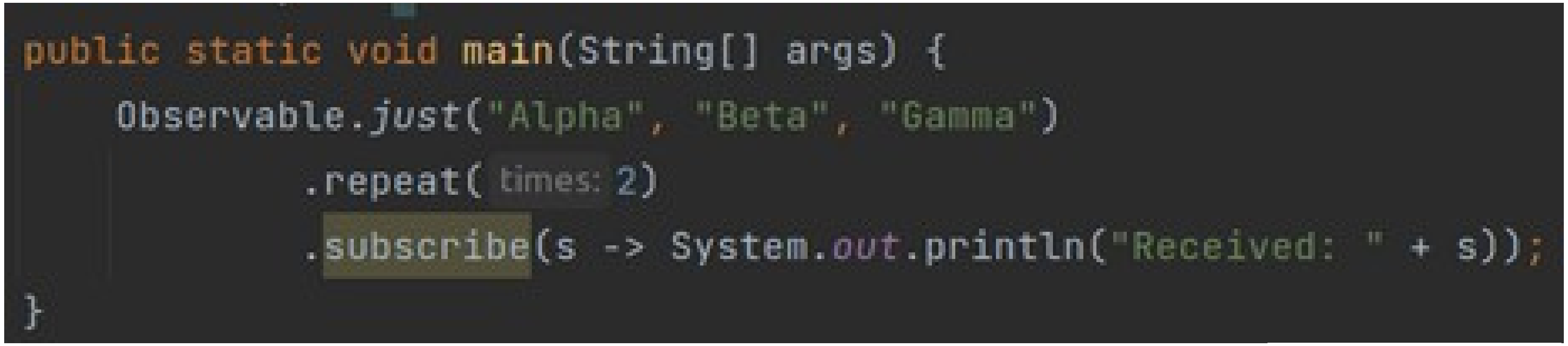
```java
public static void main(String[] args) {
    DateTimeFormatter f = DateTimeFormatter.ofPattern("MM:ss");
    System.out.println(LocalDateTime.now().format(f));
    Observable.just("Alpha", "Beta", "Gamma")
            .delay( time: 3, TimeUnit.SECONDS)
            .subscribe(s -> System.out.println(LocalDateTime.now().format(f) + " Received: " + s));
    sleep( millis: 5000);
}
```



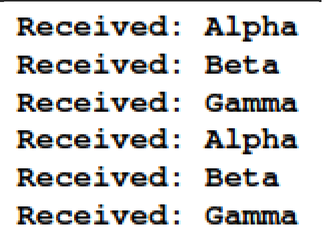
```
02:26
02:29 Received: Alpha
02:29 Received: Beta
02:29 Received: Gamma
```

# Operators:utility

- **repeat**
  - The repeat() operator will repeat subscription after onComplete() a specified number of times.



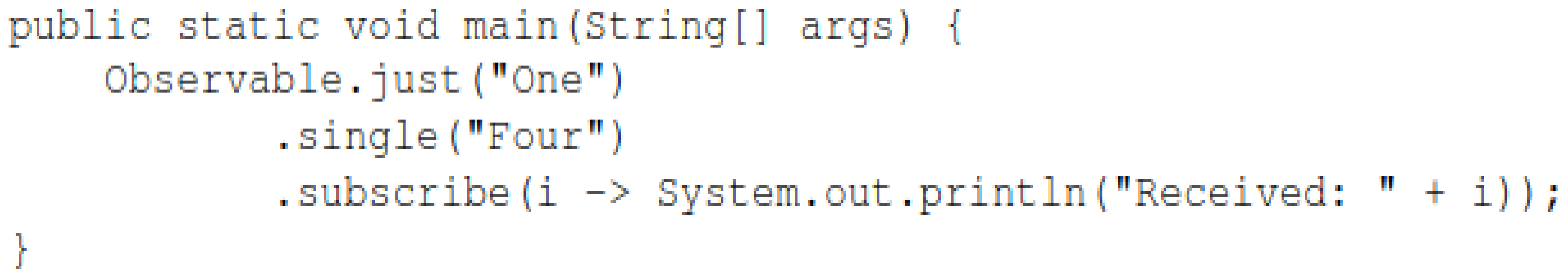
```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .repeat( times: 2)
            .subscribe(s -> System.out.println("Received: " + s));
}
```



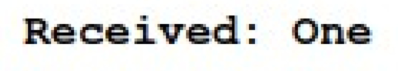
```
Received: Alpha
Received: Beta
Received: Gamma
Received: Alpha
Received: Beta
Received: Gamma
```

# Operators:utility

- **single**
  - The single() operator returns a Single that emits the item emitted by this Observable. If the Observable emits more than one item, the single() operator throws an exception.



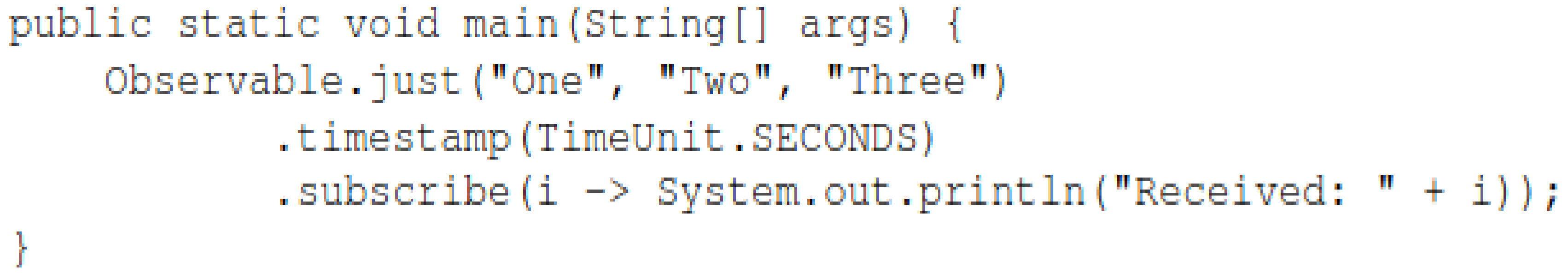
```
public static void main(String[] args) {
    Observable.just("One")
            .single("Four")
            .subscribe(i -> System.out.println("Received: " + i));
}
```



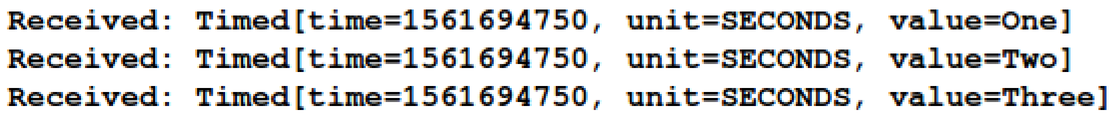
```
Received: One
```

# Operators:utility

- **timestamp**
  - The timestamp() operator attaches a timestamp to every item emitted by an Observable.



```java
public static void main(String[] args) {
    Observable.just("One", "Two", "Three")
            .timestamp(TimeUnit.SECONDS)
            .subscribe(i -> System.out.println("Received: " + i));
}
```



```
Received: Timed[time=1561694750, unit=SECONDS, value=One]
Received: Timed[time=1561694750, unit=SECONDS, value=Two]
Received: Timed[time=1561694750, unit=SECONDS, value=Three]
```

## Operators:utility

- **timeInterval**
  - The timeInterval( ) operator emits the time lapses between the consecutive emissions of a source Observable.

```
public static void main(String[] args) {
    Observable.interval(2, TimeUnit.SECONDS)
            .doOnNext(i -> System.out.println("Emitted: " + i))
            .take(3)
            .timeInterval(TimeUnit.SECONDS)
            .subscribe(i -> System.out.println("Received: " + i));
    sleep(7000);
}
```

```
Emitted: 0
Received: Timed[time=2, unit=SECONDS, value=0]
Emitted: 1
Received: Timed[time=2, unit=SECONDS, value=1]
Emitted: 2
Received: Timed[time=2, unit=SECONDS, value=2]
```